

[第一章 PostgreSQL 13 概述](#)

[第二章 了解事务和锁](#)

[第三章 使用索引](#)

[第四章 处理高级SQL](#)

[第五章 日志文件和系统统计信息](#)

[第六章 优化查询以获得良好的性能](#)

[第七章 编写存储过程](#)

[第八章 管理PG安全](#)

[第九章 处理备份和恢复](#)

[第十章 理解备份和复制](#)

[第十一章 决定有用的扩展](#)

[第十二章 排除PostgreSQL的故障](#)

[第十三章 迁移到 PostgreSQL](#)

PostgreSQL 13 概述

1. PostgreSQL 13的新功能是什么？

- 1.1 挖掘SQL和开发人员相关的话题
 - 1.1.1 改进psql的命令行处理
 - 1.1.2 改进pgbench
 - 1.1.3 更轻松地生成随机 UUID
 - 1.1.4 更快地删除数据库
 - 1.1.5 添加ALTER TABLE ... DROP EXPRESSION...
- 1.2 利用性能的改进
 - 1.2.1 B树索引去重
 - 1.3 增加增量排序的功能
 - 1.4 将 -j 8 添加到 reindexdb
 - 1.5 允许哈希聚合溢出到磁盘
 - 1.6 加速 PL/pgSQL
 - 1.7 并行化 VACUUM 操作
 - 1.8 允许跳过 WAL 进行全表写入
 - 1.9 其他性能改进
 - 1.10 让监控更强大
 - 1.11 其他系统视图

2.总结

PostgreSQL已经走过了漫长的道路。30多年来的不断发展为我们提供了一个最卓越的开源产品。多年来，PostgreSQL已经变得越来越流行。PostgreSQL 13提供了更多的功能，将给解决方案带来比以往更大的推动力。许多新功能为未来的进一步发展打开了大门，并将使开发者在未来几十年内实现最先进的技术。在这一章中，你将被介绍到这些新的功能，并将得到一个概览，了解哪些功能得到了改进、增加，甚至改变。

将涵盖以下主题。

- PostgreSQL 13的新功能是什么？
- SQL和开发人员相关的特性
- 备份、恢复和复制
- 性能相关的主题
- 存储器相关的主题

所有相关的特性都将被涵盖。当然，总是有更多的内容，还有数以千计的其他微小变化已经进入PostgreSQL 13。在本章中你将看到的是新版本的亮点。

1. PostgreSQL 13的新功能是什么？

PostgreSQL 13是一个重要的里程碑，许多大大小小的功能这次都进入了数据库核心。在本章中，你将被介绍到PostgreSQL世界中最最重要的发展。让我们开始吧，看看开发者们都想出了什么。

1.1 挖掘SQL和开发人员相关的话题

PostgreSQL 13提供了一些对开发者特别重要的新功能。

1.1.1 改进psql的命令行处理

在PostgreSQL 13中，已经提交了一个小的扩展，基本上可以帮助更好地跟踪psql中的会话状态。现在你可以看到一个事务是否在成功运行。让我们看一下一个简单的列表。

```
test=# BEGIN;
BEGIN
test=*# SELECT 1;
?column?

-----
1
(1 row)
test=*# SELECT 1 / 0;
ERROR: division by zero
test!=# SELECT 1 / 0;
ERROR: current transaction is aborted, commands ignored until end of transaction
block
test!=# COMMIT;
ROLLBACK
```

这个代码块有什么特别之处？需要注意的是，提示符已经改变。原因是%*x*已经被添加到PROMPT1和PROMPT2中。我们现在可以立即看到一个事务是否正在进行，是否处于困境，或者是否没有事务正在进行。这使得命令行处理更容易一些，而且希望能减少错误的发生。

1.1.2 改进pgbench

pgbench是最常用的PostgreSQL数据库基准测试工具之一。在旧版本中，pgbench创建了一些标准表。随着PostgreSQL 13的引入，默认的数据集现在可以被分区（开箱即用）。

下面是发生的情况。

```
% pgbench -i -s 100 --partitions=10
...
done in 19.70 s (drop tables 0.00 s, create tables 0.03 s, generate 7.34 s,
vacuum
10.24 s, primary keys 2.08 s).
test=# \d+
List of relations
 Schema |        Name        |          Type          | ...
-----+-----+-----+ ...
public | pgbench_accounts | partitioned TABLE | ...
public | pgbench_accounts_1 | TABLE | ...
public | pgbench_accounts_10 | TABLE | ...
public | pgbench_accounts_2 | TABLE | ...
public | pgbench_accounts_3 | TABLE | ...
...
```

-- 分区告诉pgbench要创建多少个分区。现在可以更容易地对一个分区的表和一个非分区的默认数据集进行基准测试。

1.1.3 更轻松地生成随机 UUID

在旧版本中，我们必须加载一个扩展来处理UUIDs。许多用户不知道这些扩展实际上是存在的，或者由于某种原因不想启用它们。在PostgreSQL 13中，不需要任何扩展就可以轻松地生成随机UUID。下面是它的工作原理。

```
test=# SELECT gen_random_uuid();
gen_random_uuid
-----
ca1754dc-b5d5-442b-a40b-9c6a9d51a9a1
(1 row)
```

1.1.4 更快地删除数据库

更快地删除数据库的能力是一个重要的功能，绝对是我最喜欢的新功能之一。问题是什么呢？

```
postgres=# DROP DATABASE test;
ERROR: database "test" is being accessed by other users
DETAIL: There is 1 other session using the database.
```

一个数据库只有在没有其他人连接到该数据库时才能被放弃。对于许多用户来说，这很难实现。通常情况下，新的连接不断出现，运行ALTER DATABASE来阻止新的连接并放弃现有的连接以杀死数据库，这实在是太麻烦了。从终端用户的角度来看，WITH（强制）选项极大地简化了事情。

```
postgres=# DROP DATABASE test WITH (force);
DROP DATABASE
```

无论系统中发生了什么，数据库都会被丢弃，这使得这个过程更加可靠。

1.1.5 添加ALTER TABLE ... DROP EXPRESSION...

PostgreSQL有能力将一个表达式的输出物化。下面的列表显示了如何使用一个生成的列。

```
test=# CREATE TABLE t_test (
  a int,
  b int,
  c int GENERATED ALWAYS AS (a * b) STORED
);
CREATE TABLE
test=# INSERT INTO t_test (a, b) VALUES (10, 20);
INSERT 0 1
```

如您所见，表格中添加了一行：

```
test=# SELECT * FROM t_test;
 a | b | c
-----+
 10 | 20 | 200
(1 row)
```

现在的问题是：我们怎样才能再次摆脱生成的表达式？PostgreSQL 13有了答案。

```
test=# ALTER TABLE t_test ALTER COLUMN c DROP EXPRESSION ;  
ALTER TABLE
```

c 现在是一个普通的列，就像所有其他列一样：

```
test=# \d t_test;  
Table "public.t_test"  
Column | Type | Collation | Nullable | Default  
-----+-----+-----+-----+  
a | integer | | |  
b | integer | | |  
c | integer | | |
```

现在你可以直接插入到c中，而不会出现错误。

1.2 利用性能的改进

PostgreSQL 13增加了一些性能改进。在这一节中，你将了解到一些更相关的改进，这些改进可以在实际生活中产生作用。当然，这里有成百上千的小改动，但本章的目的实际上是集中在能产生真正变化的最重要的方面。

1.2.1 B树索引去重

B-树是迄今为止PostgreSQL世界中最重要的索引结构。大多数的索引都是B-树。在PostgreSQL 13中，B-树得到了很大的改进。在旧版本中，相同的条目被单独存储在索引中。如果你有100万个hans的表项，索引实际上在数据结构中也会有100万个 "hans" 的副本。在PostgreSQL 12中，典型的索引条目曾经是" (value, ctid) "。这一点已经被改变了。现在PostgreSQL将更有效地处理重复的内容。

让我们创建一个例子，看看这是如何工作的。

```
test=# CREATE TABLE tab (a int, b int);  
CREATE TABLE  
test=# INSERT INTO tab SELECT id, 1 FROM generate_series(1, 5000000) AS id;  
INSERT 0 5000000
```

在这种情况下，我们已经创建了500万行。a列有500万个不同的值，而b列包含500万个相同的条目。让我们创建两个索引，看看会发生什么。

```
test=# CREATE INDEX idx_a ON tab (a);  
CREATE INDEX  
test=# CREATE INDEX idx_b ON tab (b);  
CREATE INDEX
```

表本身的大小在PostgreSQL 13中没有变化。我们将看到和以前一样的大小。

```
test=# \d+  
List of relations  
Schema | Name | Type | Owner | Persistence | Size | Description  
-----+-----+-----+-----+-----+-----+  
public | tab | table | hs | permanent | 173 MB |  
(1 row)
```

然而，重要的是，idx_b索引的大小将与idx_a索引的大小不同。

```
test=# \di+
List of relations
 Schema | Name | Type | Owner | Table | Persistence | Size | Description
-----+-----+-----+-----+-----+-----+-----+-----+
 public | idx_a | index | hs | tab | permanent | 107 MB |
 public | idx_b | index | hs | tab | permanent | 33 MB |
(2 rows)
```

正如你所看到的，PostgreSQL 13的效率要高得多，并且提供了一个更低的存储空间占用。较低的空间消耗将自动转化为较高的缓存命中率，因此，一般来说，性能更好。

如果你正在从旧版本迁移到PostgreSQL 13，考虑重新创建包含许多重复的索引，以确保你能利用这些改进。

1.3 增加增量排序的功能

PostgreSQL 13围绕索引和排序又有一个改进。新版本提供了增量排序。这意味着什么呢？假设我们有两列：a和b。在我们的例子中，a已经被排序了，因为它有索引。如果我们有一个预排序的列表，只是想增加一些列呢？在这种情况下，增量排序就会发生。为了说明这一点，让我们先放弃idx_b的索引。

```
test=# DROP INDEX idx_b;
DROP INDEX
```

我们只剩下一个索引 (idx_a)，它提供了一个按a排序的列表。下面的查询想要一个按a和b排序的结果。

```
test=# explain SELECT * FROM tab ORDER BY a, b;
QUERY PLAN
-----
 Incremental Sort (cost=0.47..376979.43 rows=5000000 width=8)
  Sort Key: a, b
  Presorted Key: a
   -> Index Scan using idx_a on tab (cost=0.43..151979.43 rows=5000000 width=8)
(4 rows)
```

PostgreSQL将进行由idx_a索引提供的增量排序。在我的机器上的总运行时间大约是1.4秒（使用默认配置）。

```
test=# explain analyze SELECT * FROM tab ORDER BY a, b;
QUERY PLAN
-----
 Incremental Sort (cost=0.47..376979.43 rows=5000000 width=8)
 (actual time=0.167..1297.696 rows=5000000 loops=1)
  Sort Key: a, b
  Presorted Key: a
  Full-sort Groups: 156250 Sort Method: quicksort
  Average Memory: 26kB Peak Memory: 26kB
   -> Index Scan using idx_a on tab (cost=0.43..151979.43 rows=5000000 width=8)
 (actual time=0.069..773.260 rows=5000000 loops=1)
 Planning Time: 0.068 ms
 Execution Time: 1437.362 ms
(7 rows)
```

为了看看在PostgreSQL 12和之前会发生什么，我们可以暂时关闭增量排序。

```
test=# SET enable_incremental_sort TO off;
SET
test=# explain analyze SELECT * FROM tab ORDER BY a, b;
QUERY PLAN
-----
Sort (cost=765185.42..777685.42 rows=5000000 width=8)
(actual time=1269.250..1705.754 rows=5000000 loops=1)
Sort Key: a, b
Sort Method: external merge Disk: 91200kB
-> Seq Scan on tab (cost=0.00..72124.00 rows=5000000 width=8)
(actual time=0.032..311.777 rows=5000000 loops=1)
Planning Time: 0.057 ms
Execution Time: 1849.416 ms
(6 rows)
```

PostgreSQL将退回到旧的执行计划，这当然需要超过400毫秒的时间，这是相当可观的。但还有一点：内存占用率更低。旧的机制必须对整个数据集进行排序--新的机制几乎不需要任何内存可言。换句话来说：增量排序不仅仅是一个性能问题。它还大大降低了内存消耗。

1.4 将-j 8 添加到 reindexdb

reindexdb命令已经存在了很多年。它可以被用来轻松地重新索引整个数据库。在PostgreSQL 13中，reindexdb得到了扩展。它现在支持-j标志，这使得你可以一次使用超过一个CPU核心来重新索引一个数据库。

```
$ reindexdb --help
reindexdb reindexes a PostgreSQL database.
Usage:
  reindexdb [OPTION]... [DBNAME]
Options:
  -a, --all    reindex all databases
  --concurrently reindex concurrently
  -d, --dbname=DBNAME database to reindex
  -e, --echo   show the commands being sent to the server
  -i, --index=INDEX recreate specific index(es) only
  -j, --jobs=NUM use this many concurrent connections to reindex
  ...
```

如接下来的列表所示，性能差异可能是相当大的。

```
$ time reindexdb -j 8 database_name
real 0m6.789s
user 0m0.012s
sys 0m0.008s
$ time reindexdb database_name
real 0m24.137s
user 0m0.001s
sys 0m0.004s
```

在这种情况下，为了加快进程，使用了8个核心。请记住，只有当你有足够的索引可以同时创建时，`-j`选项才会加速事情。如果你只有一个表和一个索引，就不会有任何改善。你有越多的大表，结果就会越好。

1.5 允许哈希聚合溢出到磁盘

当运行简单的GROUP BY语句时，PostgreSQL基本上有两个选项来执行这些类型的查询。

- 组聚合
- 哈希聚合

当组的数量真的很大时，通常会使用组的聚合。如果你按电话号码对数十亿人进行分组，那么组的数量就很高，PostgreSQL无法在内存中完成。组聚合是执行这种类型查询的首选方法。第二个选择是散列聚合。假设你按性别对数十亿人进行分组。结果组的数量会很小，因为根本就只有少量的性别。然而，如果规划者弄错了呢？如果规划器对组的数量产生了一个错误的估计呢？从PostgreSQL 13开始，如果一个哈希聚合所使用的内存超过`work_mem`，就有可能使哈希聚合的数据溢出到磁盘。为了看看会发生什么，我将首先运行一个简单的GROUP BY语句。

```
test=# explain SELECT a, count(*) FROM tab GROUP BY 1;
QUERY PLAN
-----
GroupAggregate (cost=0.43..204852.43 rows=5000000 width=12)
Group Key: a
-> Index Only Scan using idx_a on tab (cost=0.43..129852.43 rows=5000000
width=4)
(3 rows)
```

PostgreSQL预计组的数量是500万行，并去做组的聚合。但是，如果我们让索引变得更昂贵(`random_page_cost`)，并告诉PostgreSQL真的有足够的内存来处理这个操作呢？让我们来试试。

```
test=# SET random_page_cost TO 100;
SET
test=# SET work_mem TO '10 GB';
SET
test=# explain SELECT a, count(*) FROM tab GROUP BY 1;
QUERY PLAN
-----
HashAggregate (cost=97124.00..147124.00 rows=5000000 width=12)
Group Key: a
-> Seq Scan on tab (cost=0.00..72124.00 rows=5000000 width=4)
(3 rows)
```

规划器将决定一个哈希集合，并在内存中执行，具体如下。

```
test=# explain analyze SELECT a, count(*) FROM tab GROUP BY 1;
QUERY PLAN
-----
HashAggregate (cost=97124.00..147124.00 rows=5000000 width=12)
(actual time=2466.159..3647.708 rows=5000000 loops=1)
Group Key: a
Peak Memory Usage: 589841 kB
-> Seq Scan on tab (cost=0.00..72124.00 rows=5000000 width=4)
(actual time=0.061..615.607 rows=5000000 loops=1)
Planning Time: 0.065 ms
Execution Time: 3868.609 ms
(6 rows)
```

正如你所看到的，系统使用了大约600MB的内存（在峰值消耗时）来运行该操作。但是，如果我们把 `work_mem` 设置为刚好低于这个值，会发生什么？让我们来了解一下。

```
test=# SET work_mem TO '500 MB';
SET
test=# explain analyze SELECT a, count(*) FROM tab GROUP BY 1;
QUERY PLAN
-----
HashAggregate (cost=97124.00..147124.00 rows=5000000 width=12)
(actual time=2207.005..3778.903 rows=5000000 loops=1)
Group Key: a
Peak Memory Usage: 516145 kB Disk Usage: 24320 kB
HashAgg Batches: 4
-> Seq Scan on tab (cost=0.00..72124.00 rows=5000000 width=4)
(actual time=0.054..545.678 rows=5000000 loops=1)
Planning Time: 0.092 ms
Execution Time: 3970.435 ms
(6 rows)
```

500MB的`work_mem`是可用的，但我们的操作会用到稍微多一点。因此，PostgreSQL将把这些额外的数据发送到磁盘。如果我们进一步减少内存，PostgreSQL将回落到一个群集。然而，在估计稍有错误的情况下，这个新功能确实能起到帮助作用。

1.6 加速 PL/pgSQL

PL/pgSQL是在PostgreSQL中运行存储过程的唯一最流行的语言。然而，PL/pgSQL的一些角落曾经相当缓慢。PostgreSQL 13具有一些性能改进，以帮助使用PL/pgSQL的开发者运行更快的代码。下面是一个例子。

```
CREATE OR REPLACE FUNCTION slow_pi() RETURNS double precision AS $$
DECLARE
a double precision := 1;
s double precision := 1;
r double precision := 0;
BEGIN
FOR i IN 1 .. 10000000 LOOP
r := r + s/a;
a := a + 2;
s := -s;
END LOOP;
```

```
RETURN 4 * r;
END;
$$ LANGUAGE plpgsql;
SELECT slow_pi();
slow_pi
-----
3.1415925535898497
(1 row)
Time: 13060.650 ms (00:13,061) vs 2108,464 ms (00:02,108)
```

我们的函数是一种计算圆周率的超慢方法。有趣的是，循环被加速了，代码的执行速度比以前快了几倍。美中不足的是，代码本身并没有被改变--事情只是在默认情况下运行得更快。

1.7 并行化 VACUUM 操作

在每一个PostgreSQL的部署中都需要VACUUM。从13版开始，PostgreSQL可以并行地处理关系上的索引。其工作方式如下。

```
VACUUM (verbose ON, analyze ON, parallel 4) some_table;
```

简单地告诉VACUUM你可能想使用多少个核心，PostgreSQL将尝试使用最终用户指定的多少个核心。

1.8 允许跳过 WAL 进行全表写入

当wal_level被设置为minimal时（现在已经不是默认值了），你将能够享受一个真正有趣的性能改进：如果一个关系（通常是表或物化视图的同义词）大于wal_skip_threshold，PostgreSQL会尝试通过直接同步关系而不是通过事务日志机制来避免写入WAL。在旧版本中，只有COPY可以做到这一点。根据你的存储的属性，如果提交一个事务的速度慢到不可接受的程度，提高或降低这个值可能会有帮助。

请记住，如果你想使用复制，wal_level = minimal是不可能的。

1.9 其他性能改进

除了我们刚才看到的变化之外，还有一些性能上的增强。第一件值得一提的事情是能够使用ALTER STATISTICS ... 设置统计数据。PostgreSQL现在允许在同一个查询中使用多个扩展的统计数据，如果你的操作相当复杂，这可能相当有用。

为了告诉PostgreSQL你的存储系统可以处理多少I/O并发，在postgresql.conf中增加了maintain_io_concurrency参数。

```
test=# SHOW maintenance_io_concurrency;
maintenance_io_concurrency
-----
10
(1 row)
```

这个参数允许管理任务以一种更巧妙的方式行事。

如果你碰巧对真正的大表进行操作，你可能会喜欢下一个功能：大型关系现在可以更快地被截断。如果你只在有限的时间内存储数据，这可能是非常有用的。

为了存储大字段（通常>1,996字节），PostgreSQL使用了一种通常被称为TOAST（超大属性存储技术）的技术。PostgreSQL改善了TOAST的解压性能，并允许更有效地检索TOAST字段中的领先字节。

传统上，你必须在Windows上使用TCP/IP连接。问题是，TCP/IP连接比简单的UNIX套接字更容易受到更高的延迟影响。现在Windows支持套接字连接，这有助于减少本地连接的延迟。

最后，LISTEN/NOTIFY也得到了改进，使之具有更好的性能。

1.10 让监控更强大

监控是每个成功的数据库操作的关键。PostgreSQL的这一领域也得到了改进。

1.11 其他系统视图

为了让管理员和DevOps工程师的生活更轻松，PostgreSQL社区在第13版中增加了一些易于使用的系统视图。

我最喜欢的一个新东西是一个系统视图，它允许我们跟踪pg_basebackup的进展。

```
test=# \d pg_stat_progress_basebackup
View "pg_catalog.pg_stat_progress_basebackup"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----+
pid | integer | |
phase | text | |
backup_total | bigint | |
backup_streamed | bigint | |
tablespaces_total | bigint | |
tablespaces_streamed | bigint | |
```

以前，很难看到备份进展到什么程度。ANALYZE的情况也是如此。在大表中，它可能需要相当长的时间，所以又增加了一个视图。

```
test=# \d pg_stat_progress_analyze
View "pg_catalog.pg_stat_progress_analyze"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----+
pid | integer | |
datid | oid | |
datname | name | |
relid | oid | |
phase | text | |
sample_blnks_total | bigint | |
sample_blnks_scanned | bigint | |
ext_stats_total | bigint | |
ext_stats_computed | bigint | |
child_tables_total | bigint | |
child_tables_done | bigint | |
current_child_table_relid | oid | |
```

pg_stat_progress_analyze告诉你关于抽样的情况，也为你提供了关于分区表的统计数据等等。

同样重要的是，pg_stat_replication中加入了字段。

```

test=# \d pg_stat_replication
View "pg_catalog.pg_stat_replication"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
pid | integer | |||
...
spill_txns | bigint | |||
spill_count | bigint | |||
spill_bytes | bigint | |||

```

如果你使用逻辑解码，可能会发生一些事务不得不进入磁盘的情况，因为没有足够的RAM可用来处理要应用于复制的待定变化。pg_stat_replication现在可以跟踪这一信息。第13版中还增加了一个视图，即pg_shmem_allocations。它允许用户跟踪分配的共享内存。

```

test=# \d pg_shmem_allocations
View "pg_catalog.pg_shmem_allocations"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
name | text | |||
off | bigint | |||
size | bigint | |||
allocated_size | bigint | |||

```

基本上，你可以看到什么东西分配了多少内存给什么东西。如果你正在处理许多不同的扩展和分配共享内存的事情（即不是本地内存，如work_mem等），这就特别有用。

最后，还有pg_stat_slru。现在的问题是：什么是SLRU？在PostgreSQL中，不仅仅是索引和表，还有更多的工作。在内部，PostgreSQL需要缓存一些东西，比如CLOG（Commit Log的缩写）、Multixacts和子事务数据。这就是SLRU缓存的作用。如果有许多不同的事务被访问，提交日志（不要与WAL（Write Ahead Log）混为一谈，它也被称为xlog）可能是一个主要的争用来源。

```

test=# \d pg_stat_slru
View "pg_catalog.pg_stat_slru"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
name | text | |||
blk_zeroed | bigint | |||
blk_hit | bigint | |||
blk_read | bigint | |||
blk_written | bigint | |||
blk_exists | bigint | |||
flushes | bigint | |||
truncates | bigint | |||
stats_reset | timestamp with time zone | |||

```

所有这些系统视图将使我们更容易看到PostgreSQL内部发生的事情。我们期望将来会有更多的视图加入。我们已经看到了几个针对PostgreSQL 14的补丁。Laurenz Albe（在Cybertec PostgreSQL International，我的公司）已经提出了一个补丁，帮助跟踪连接数和更多的东西。

2.总结

PostgreSQL 13提供了各种功能，包括性能改进，简化数据库处理，以及更多。更好的Btrees无疑是PostgreSQL的一个亮点。然而，像增量排序这样的东西也将有助于良好的性能，而额外的视图将有助于管理员有更好的可视性。我们可以期待在PostgreSQL 14中看到更多更酷的功能。特别是，围绕B树和其他性能问题的改进，以及更好的监控，将帮助最终用户更好、更有效、更容易地运行应用程序。

在下一章中，你将被介绍到事务和锁定，这对可扩展性和存储管理以及性能都很重要。

了解事务和锁

现在我们已经介绍了PostgreSQL 13的介绍，我们想把注意力集中在下一个重要的话题上。锁定对于任何类型的数据库都是一个重要的概念。仅仅了解它是如何工作的还不够，还需要写出适当的或更好的应用程序--从性能的角度来看，它也是必不可少的。如果不正确地处理锁，你的应用程序不仅可能很慢，而且还可能出现非常意外的行为。在我看来，锁是性能的关键，对它有一个很好的概述肯定会有帮助。因此，了解锁和事务对管理员和开发人员都很重要。在本章中，你将了解到以下内容。

- 使用PostgreSQL事务
- 了解基本的锁
- 利用FOR SHARE和FOR UPDATE
- 了解事务隔离级别
- 观察死锁和类似问题
- 利用咨询锁
- 优化存储和管理清理

在本章结束时，你将能够理解并以最有效的方式利用PostgreSQL事务。你将看到，许多应用程序可以从性能的提高中受益。

1. 使用PostgreSQL事务

PostgreSQL为你提供了高度先进的事务机制，为开发者和管理员提供了无数的功能。在这一节中，我们将看一下事务的基本概念。要知道的第一件重要的事情是，在PostgreSQL中，所有的东西都是一个事务。如果你向服务器发送一个简单的查询，它已经是一个事务了。下面是一个例子。

```
test=# SELECT now(), now();
now | now
+-----+
2020-08-13 11:03:17.741316+02 | 2020-08-13 11:03:17.741316+02
(1 row)
```

在这种情况下，SELECT语句将是一个单独的事务。如果再次执行相同的命令，将返回不同的时间戳。

请记住，now()函数将返回事务时间。因此，SELECT语句将总是返回两个相同的时间戳。如果你想要真实的时间，考虑使用clock_timestamp()而不是now()。

如果多个语句必须是同一事务的一部分，必须使用BEGIN语句，如下所示。

```
test=# \h BEGIN
Command: BEGIN
Description: start a transaction block
Syntax:
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]
where transaction_mode is one of:
ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
READ WRITE | READ ONLY
[ NOT ] DEFERRABLE
URL: https://www.postgresql.org/docs/13/sql-begin.html
```

BEGIN语句将确保一条以上的命令被打包到一个事务中。下面是它的工作原理。

```
test=# BEGIN;
BEGIN
test=# SELECT now();
now
-----
2020-08-13 11:04:15.379104+02
(1 row)
test=# SELECT now();
now
-----
2020-08-13 11:04:15.379104+02
(1 row)
test=# COMMIT;
COMMIT
```

这里重要的一点是，两个时间戳将是相同的。正如我们前面提到的，我们正在谈论事务时间。为了结束事务，可以使用COMMIT

```
test=# \h COMMIT
Command: COMMIT
Description: commit the current transaction
Syntax:
COMMIT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
URL: https://www.postgresql.org/docs/13/sql-commit.html
```

这里有一些语法元素。你可以只使用COMMIT，COMMIT WORK，或COMMIT TRANSACTION。这三个命令的含义都是一样的。如果这还不够，还有更多。

```
test=# \h END
Command: END
Description: commit the current transaction
Syntax:
END [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
URL: https://www.postgresql.org/docs/13/sql-end.html
```

END子句与COMMIT子句相同。

ROLLBACK是COMMIT的对应项。它不是成功地结束一个事务，而是简单地停止该事务，而不会让其他事务看到事情，如下面的代码所示。

```
test=# \h ROLLBACK
Command: ROLLBACK
Description: abort the current transaction
Syntax:
ROLLBACK [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
URL: https://www.postgresql.org/docs/13/sql-rollback.html
```

有些应用程序使用ABORT而不是ROLLBACK。其含义是一样的。在PostgreSQL 12中，新的东西是链式事务的概念。这一切的意义是什么呢？下面的列表显示了一个例子。

```
test=# SHOW transaction_read_only;
transaction_read_only

-----
off
(1 row)
test=# BEGIN TRANSACTION READ ONLY ;
BEGIN
test=*# SELECT 1;
?column?

-----
1
(1 row)
test=*# COMMIT AND CHAIN;
COMMIT
test=*# SHOW transaction_read_only;
transaction_read_only

-----
on
(1 row)
test=*# SELECT 1;
?column?

-----
1
(1 row)
test=*# COMMIT AND NO CHAIN;
COMMIT
test=# SHOW transaction_read_only;
transaction_read_only

-----
off
(1 row)
test=# COMMIT;
WARNING: there is no transaction in progress
COMMIT
```

让我们一步一步地看这个例子：

1. 显示transaction_read_only设置的内容。它是关闭的，因为在默认情况下，我们处于读/写模式。
2. 使用BEGIN启动一个只读事务。这将自动调整transaction_read_only变量。
3. 使用AND CHAIN提交事务，然后PostgreSQL会自动启动一个新的事务，其属性与之前的事务相同。

在我们的例子中，我们也将处于只读模式，就像之前的事务一样。不需要显式地打开一个新的事务并再次设置什么值，这可以大大减少应用程序和服务器之间的往返次数。如果一个事务被正常提交 (=NO CHAIN)，该事务的只读属性将消失。

1.1 处理事务中的错误

事务从开始到结束并不总是正确的。事情可能因为某种原因而出错。然而，在PostgreSQL中，只有无错误的事务才能被提交。下面的列表显示了一个失败的事务，它由于一个除以0的错误而出错。

```
test=# BEGIN;
BEGIN
test=# SELECT 1;
?column?

-----
1
(1 row)
test=# SELECT 1 / 0;
ERROR: division by zero
test!=# SELECT 1;
ERROR: current transaction is aborted, commands ignored until end of transaction
block
test!=# SELECT 1;
ERROR: current transaction is aborted, commands ignored until end of transaction
block
test!=# COMMIT;
ROLLBACK
```

请注意，除以零没有结果。

在任何适当的数据库中，类似这样的指令会立即出错，使语句失败。

需要指出的是，PostgreSQL会出错。在错误发生后，将不再接受任何指令，即使这些指令在语义和语法上是正确的。仍然有可能发出COMMIT。然而，PostgreSQL会回滚交易，因为这是当时唯一正确的做法。

1.2 使用保存点

在专业的应用程序中，要写出合理的长事务而不遇到一个错误是相当困难的。为了解决这个问题，用户可以利用称为SAVEPOINT的东西。顾名思义，保存点是事务中一个安全的地方，如果事情出了大错，应用程序可以返回。下面是一个例子。

```
test=# BEGIN;
BEGIN
test=# SELECT 1;
?column?

-----
1
(1 row)
test=# SAVEPOINT a;
SAVEPOINT
test=# SELECT 2 / 0;
ERROR: division by zero
test!=# SELECT 2;
ERROR: current transaction is aborted, commands ignored until end of transaction
block
test!=# ROLLBACK TO SAVEPOINT a;
```

```
ROLLBACK
test=# SELECT 3;
?column?
-----
3
(1 row)
test=# COMMIT;
COMMIT
```

在第一个SELECT子句之后，我决定创建一个保存点，以确保应用程序能够始终返回到事务内部的这一点。正如你所看到的，这个保存点有一个名字，这个名字将在后面提到。

返回到名为a的保存点后，事务可以正常进行。代码已经跳回到了错误之前，所以一切都很正常。

一个事务内的保存点的数量实际上是无限的。我们已经看到客户在一次操作中拥有超过25万个保存点。PostgreSQL可以很容易地处理这个问题。

如果你想从一个事务内部删除一个保存点，有一个RELEASE SAVEPOINT命令。

```
test=# \h RELEASE
Command: RELEASE SAVEPOINT
Description: destroy a previously defined savepoint
Syntax:
RELEASE [ SAVEPOINT ] savepoint_name
URL: https://www.postgresql.org/docs/13/sql-release-savepoint.html
```

许多人问，如果你在事务结束后试图到达一个保存点，会发生什么？答案是，事务一结束，保存点的生命就会结束。换句话说，在交易完成后，没有办法返回到某一时间点。

1.3 事务性的DDLS

PostgreSQL有一个非常好的功能，不幸的是，这个功能在许多商业数据库系统中是不存在的。在PostgreSQL中，可以在一个事务块中运行DDL（改变数据结构的命令）。在一个典型的商业系统中，一个DDL将隐含地提交当前的事务。这在PostgreSQL中不会发生。

除了一些小的例外（DROP DATABASE、CREATE TABLESPACE、DROP TABLESPACE等等），PostgreSQL中的所有DDL都是事务性的，这是一个巨大的优势，对终端用户是一个真正的好处。

下面是一个例子。

```
test=# BEGIN;
BEGIN
test=# CREATE TABLE t_test (id int);
CREATE TABLE
test=# ALTER TABLE t_test ALTER COLUMN id TYPE int8;
ALTER TABLE
test=# \d t_test
Table "public.t_test"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
id | bigint | | |
test=# ROLLBACK;
ROLLBACK
test=# \d t_test
Did not find any relation named "t_test".
```

在这个例子中，一个表被创建和修改了，整个事务被中止了。正如你所看到的，没有隐含的COMMIT命令或任何其他奇怪的行为。PostgreSQL只是按照预期的方式行事。

如果你想部署软件，事务性DDL就特别重要。试想一下，运行一个内容管理系统（CMS）。如果一个新的版本发布了，你会想要升级。运行旧版本仍然是可以的；运行新版本也是可以的，但你真的不希望出现新旧混合的情况。因此，在一个事务中部署升级是非常有利的，因为它是一个原子性的升级操作。

为了促进良好的软件实践，我们可以将源码控制系统中几个单独编码的模块纳入一个部署事务。

2.了解基本的锁

在本节中，你将学习基本的锁定机制。其目的是了解锁的一般工作原理，以及如何正确地进行简单的应用。

为了向你展示事情是如何进行的，我们将创建一个简单的表。出于演示的目的，我将使用一个简单的INSERT命令向表中添加一条记录。

```
test=# CREATE TABLE t_test (id int);
CREATE TABLE
test=# INSERT INTO t_test VALUES (0);
INSERT 0 1
```

第一件重要的事情是，表可以被并发地读取。许多用户在同一时间读取相同的数据，不会互相阻塞。这使得PostgreSQL能够处理成千上万的用户而没有任何问题。

现在的问题是，如果读和写同时发生会怎样？这里有一个例子。让我们假设该表包含一条记录，其id=0。

Transaction 1	Transaction 2
BEGIN;	BEGIN;
UPDATE t_test SET id = id + 1 RETURNING *;	
User will see 1	SELECT * FROM t_test;
	User will see 0
COMMIT;	COMMIT;

有两个事务被打开。第一个将改变一行。然而，这并不是一个问题，因为第二个事务可以继续进行。它将返回UPDATE之前的旧行。这种行为被称为多版本并发控制（MVCC）。

一个事务只有在写事务在读事务启动之前已经提交的情况下才会看到数据。一个事务不能检查另一个活动连接所做的改变。一个事务只能看到那些已经被提交的变化

还有第二个重要的方面--许多商业或开源数据库仍然无法处理并发的读和写。在PostgreSQL中，这绝对不是一个问题--读和写可以并存。

写事务不会阻塞读事务。

在事务提交后，该表将包含1.如果两个人同时改变数据会发生什么？下面是一个例子。

Transaction 1	Transaction 2
BEGIN;	BEGIN;
UPDATE t_test SET id = id + 1 RETURNING *;	
It will return 2	UPDATE t_test SET id = id + 1 RETURNING *;
	It will wait for transaction 1
COMMIT;	It will wait for transaction 1
	It will reread the row, find 2, set the value, and return 3
	COMMIT;

假设你想计算一个网站的点击次数。如果你运行前面的代码，不会有任何点击率丢失，因为PostgreSQL保证一个UPDATE语句在另一个之后执行。

PostgreSQL只锁定受UPDATE影响的行。所以，如果你有1000条记录，理论上你可以在同一个表上运行1000个并发变化。

同样值得注意的是，你总是可以运行并发的读取。我们的两个写不会阻塞读。

2.1 避免典型错误和显式锁定

在我作为一个专业的PostgreSQL顾问（<https://www.cybertec-postgresql.com>）的生活中，我看到有几个错误经常被重复。如果说生活中存在常数，这些典型的错误绝对是一些永远不会改变的东西。

这是我最喜欢的：

Transaction 1	Transaction 2
BEGIN;	BEGIN;
SELECT max(id) FROM product;	SELECT max(id) FROM product;
User will see 17	User will see 17
User will decide to use 18	User will decide to use 18
INSERT INTO product ... VALUES (18, ...)	INSERT INTO product ... VALUES (18, ...)
COMMIT;	COMMIT;

在这种情况下，要么有一个重复的密钥违规，要么有两个相同的条目。这两个问题的变化都不那么吸引人。解决这个问题的一个方法是使用显式表锁。下面的代码向我们展示了LOCK的语法定义。

```
test=# \h LOCK
Command: LOCK
Description: lock a table
Syntax:
LOCK [ TABLE ] [ ONLY ] name [ * ] [ , ... ] [ IN lockmode MODE ] [ NOWAIT ]
where lockmode is one of:
    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
    | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
URL: https://www.postgresql.org/docs/13/sql-lock.html
```

正如你所看到的，PostgreSQL提供了八种类型的锁来锁定整个表。在PostgreSQL中，锁可以像ACCESS SHARE锁一样轻，也可以像ACCESS EXCLUSIVE锁一样重。下面的列表显示了这些锁的作用。

- ACCESS SHARE：这种类型的锁被读取，只和ACCESS EXCLUSIVE冲突，ACCESS EXCLUSIVE是由DROP TABLE等设置的。实际上，这意味着如果一个表即将被丢弃，SELECT就不能启动。这也意味着DROP TABLE必须要等到读取事务完成后才能开始。
- ROW SHARE：PostgreSQL在SELECT FOR UPDATE/SELECT FOR SHARE的情况下使用这种锁。它与EXCLUSIVE和ACCESS EXCLUSIVE冲突。
- ROW EXCLUSIVE：这种锁由INSERT, UPDATE, 和 DELETE 使用。它与SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE冲突。
- SHARE UPDATE EXCLUSIVE：这种锁被CREATE INDEX CONCURRENTLY, ANALYZE, ALTER TABLE, VALIDATE, 和一些其他类型的ALTER TABLE, 以及VACUUM (不是VACUUM FULL) 使用。它与SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式相冲突。
- SHARE：当索引被创建时，SHARE锁将被设置。它与ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE冲突。
- SHARE ROW EXCLUSIVE：这是由CREATE TRIGGER和某些形式的ALTER TABLE设置的，与ACCESS SHARE以外的所有东西冲突。
- EXCLUSIVE：这种类型的锁是迄今为止限制性最强的一种。它可以防止读和写。如果这个锁被一个事务占用了，其他人就不能对被影响的表进行读写。
- ACCESS EXCLUSIVE：这种锁可以防止并发的事务读和写。

考虑到PostgreSQL的锁定基础设施，我们之前概述的最大问题的一个解决方案是如下的。下面代码中的例子显示了如何锁定一个表。

```
BEGIN;
LOCK TABLE product IN ACCESS EXCLUSIVE MODE;
INSERT INTO product SELECT max(id) + 1, ... FROM product;
COMMIT;
```

请记住，这是一种相当讨厌的操作方式，因为在你的操作过程中，没有其他人可以读或写到表。因此，应该不惜一切代价避免使用ACCESS EXCLUSIVE。

2.2 考虑替代解决方案

对于这个问题，有一个替代的解决方案。考虑一个例子，你被要求编写一个应用程序来生成发票号码。税务局可能要求你创建没有空白和无重复的发票号码。你将如何做到这一点？当然，一个解决方案是使用表锁。然而，你真的可以做得更好。下面是你可以做的，来处理我们要解决的编号问题。

```
test=# CREATE TABLE t_invoice (id int PRIMARY KEY);
CREATE TABLE
test=# CREATE TABLE t_watermark (id int);
CREATE TABLE
test=# INSERT INTO t_watermark VALUES (0);
INSERT 0
test=# WITH x AS (UPDATE t_watermark SET id = id + 1 RETURNING *)
  INSERT INTO t_invoice
    SELECT * FROM x RETURNING *;
id
-----
1
(1 row)
```

在这种情况下，我们引入了一个名为t_watermark的表。它只包含一条记录。首先将执行WITH命令。该行将被锁定和递增，并返回新的值。每次只有一个人可以做这个事情。CTE返回的值会在t_invoice表中使用。它被保证是唯一的。美中不足的是，在t_watermark表上只有一个简单的行锁，这导致发票表中没有读取被阻止。总的来说，这种方式更具可扩展性。

3.利用FOR SHARE和FOR UPDATE

有时，从数据库中选择数据，然后在应用程序中进行一些处理，最后，在数据库中进行一些修改。这是一个典型的SELECT FOR UPDATE的例子。

下面是一个例子，显示了SELECT经常以错误的方式执行。

```
BEGIN;
SELECT * FROM invoice WHERE processed = false;
** application magic will happen here ***
UPDATE invoice SET processed = true ...
COMMIT;
```

这里的问题是，两个人可能会选择相同的未经处理的数据。对这些处理过的行所做的修改就会被覆盖掉。简而言之，将发生一个竞赛条件。

为了解决这个问题，开发人员可以利用SELECT FOR UPDATE。下面是它的使用方法。下面的例子将展示一个典型的场景。

```
BEGIN;
SELECT * FROM invoice WHERE processed = false FOR UPDATE;
** application magic will happen here ***
UPDATE invoice SET processed = true ...
COMMIT;
```

SELECT FOR UPDATE将像UPDATE一样锁定记录。这意味着没有变化可以同时发生。所有的锁都会像往常一样在COMMIT时被释放。

如果一个SELECT FOR UPDATE命令正在等待另一个SELECT FOR UPDATE命令，你将不得不等待，直到另一个命令完成（COMMIT或ROLLBACK）。如果第一个事务不想结束，不管什么原因，第二个事务有可能永远等待。为了避免这种情况，可以使用SELECT FOR UPDATE NOWAIT。

Transaction 1	Transaction 2
BEGIN;	BEGIN;
SELECT ... FROM tab WHERE ... FOR UPDATE NOWAIT;	
Some processing	SELECT ... FROM tab WHERE ... FOR UPDATE NOWAIT;
Some processing	ERROR: could not obtain lock on row in relation tab

如果NOWAIT对你来说不够灵活，可以考虑使用lock_timeout。它将包含你想在锁上等待的时间量。你可以在每个会话级别上设置它。

```
test=# SET lock_timeout TO 5000;
SET
```

在这种情况下，该值被设置为5秒。

虽然SELECT基本上不做锁定，但SELECT FOR UPDATE却可以很苛刻。试想一下下面的业务流程：我们想让一架有200个座位的飞机满员。许多人想同时预订座位。在这种情况下，可能会发生以下情况。

Transaction 1	Transaction 2
BEGIN;	BEGIN;
SELECT ... FROM flight LIMIT 1 FOR UPDATE;	
Waiting for user input	SELECT ... FROM flight LIMIT 1 FOR UPDATE;
Waiting for user input	It has to wait

问题是，每次只能预订一个座位。有可能有200个座位，但每个人都必须等待第一个人。当第一个座位被封锁的时候，其他人就不能预订座位了，即使人们并不关心最后得到哪个座位。

SELECT FOR UPDATE SKIP LOCKED将解决这个问题。让我们先创建一些样本数据。

```
test=# CREATE TABLE t_flight AS
    SELECT * FROM generate_series(1, 200) AS id;
SELECT 200
```

现在，神奇的事情来了。

Transaction 1	Transaction 2
BEGIN;	BEGIN;
SELECT * FROM t_flight LIMIT 2 FOR UPDATE SKIP LOCKED;	SELECT * FROM t_flight LIMIT 2 FOR UPDATE SKIP LOCKED;
It will return 1 and 2	It will return 3 and 4

如果每个人都想获取两行数据，我们就可以同时为100个并发事务提供服务，而不必担心事务阻塞的问题。

请记住，等待是最慢的执行方式。如果一次只能有一个事务处于活动状态，那么如果你的真正问题是由一般的锁定和冲突事务引起的，那么购买更昂贵的服务器是没有意义的。

然而，还有一点。在某些情况下，FOR UPDATE会产生意想不到的后果。大多数人没有意识到FOR UPDATE会对外键产生影响的事实。让我们假设我们有两个表：一个用于存储货币，另一个用于存储账户。下面的代码显示了这方面的一个例子。

```
CREATE TABLE t_currency (id int, name text, PRIMARY KEY (id));
INSERT INTO t_currency VALUES (1, 'EUR');
INSERT INTO t_currency VALUES (2, 'USD');
CREATE TABLE t_account (
    id int,
    currency_id int REFERENCES t_currency (id)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
    balance numeric);
INSERT INTO t_account VALUES (1, 1, 100);
INSERT INTO t_account VALUES (2, 1, 200);
```

现在，我们要在帐户表上运行 SELECT FOR UPDATE：

Transaction 1	Transaction 2
BEGIN;	
SELECT * FROM t_account FOR UPDATE;	BEGIN;
Waiting for user to proceed	UPDATE t_currency SET id = id * 10;
Waiting for user to proceed	It will wait on transaction 1

虽然在账户上有一个SELECT FOR UPDATE命令，但货币表的UPDATE命令将被阻止。这是必要的，因为，否则就有可能完全破坏外键约束。因此，在一个相当复杂的数据结构中，你很容易在一个最不希望出现的区域（一些非常重要的查询表）出现争执。

除了FOR UPDATE之外，还有FOR SHARE、FOR NO KEY UPDATE和FOR KEY SHARE。下面的列表描述了这些模式的实际含义。

- FOR NO KEY UPDATE：这个和FOR UPDATE很相似。然而，锁的作用较弱，因此，它可以与SELECT FOR SHARE共存。
- FOR SHARE：FOR UPDATE是非常强大的，它的工作假设是你肯定会改变行。FOR SHARE则不同，因为不止一个事务可以同时持有FOR SHARE锁。
- FOR KEY SHARE：它的行为类似于FOR SHARE，只是锁的作用比较弱。它将阻止FOR UPDATE，但不会阻止FOR NO KEY UPDATE。

这里最重要的是简单地尝试一下，观察一下会发生什么。改善锁定行为真的很重要，因为它可以极大地提高你的应用程序的可扩展性。

4.了解事务隔离级别

到现在为止，你已经看到了如何处理锁，以及一些基本的并发性。在本节中，你将学习事务隔离。对我来说，这是现代软件开发中最被忽视的话题之一。只有一小部分软件开发者真正意识到了这个问题，这反过来又导致了令人匪夷所思的错误。

下面是一个可能发生的例子。

Transaction 1	Transaction 2
BEGIN;	
SELECT sum(balance) FROM t_account;	
User will see 300	BEGIN;
	INSERT INTO t_account (balance) VALUES (100);
	COMMIT;
SELECT sum(balance) FROM t_account;	
User will see 400	
COMMIT;	

大多数用户实际上会期望第一笔事务总是返回300，而不管第二笔事务如何。然而，这并不正确。默认情况下，PostgreSQL运行在READ COMMITTED事务隔离模式下。这意味着事务内的每个语句都会得到一个新的数据快照，这个快照在整个查询过程中是不变的。

一个SQL语句将对同一快照进行操作，并在运行时忽略并发事务的变化。

如果你想避免这种情况，你可以使用 TRANSACTION ISOLATION LEVEL REPEATABLE READ。在这个事务隔离级别中，一个事务将在整个事务中使用同一个快照。下面是将发生的情况。

Transaction 1	Transaction 2
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;	
SELECT sum(balance) FROM t_account;	
User will see 300	BEGIN;
	INSERT INTO t_account (balance) VALUES (100);
	COMMIT;
SELECT sum(balance) FROM t_account;	SELECT sum(balance) FROM t_account;
User will see 300	User will see 400
COMMIT;	

正如我们所概述的，第一笔事务将冻结其数据的快照，并在整个事务过程中为我们提供恒定的结果。如果你想运行报告，这个功能尤其重要。报告的第一页和最后一页应该始终是一致的，并在相同的数据上操作。因此，可重复读取是一致性报告的关键。请注意，与隔离有关的错误不会总是即时出现。有时，麻烦是在一个应用程序被转移到生产中多年后才被注意到的。

可重复读取并不比读取承诺的费用高。没有必要担心性能上的损失。对于正常的在线事务处理(OLTP)，读提交有各种优势，因为可以更早地看到变化，而且发生意外错误的几率通常更低。

4.1 考虑序列化快照隔离

在读承诺和可重复读的基础上，PostgreSQL提供了可序列化的快照隔离（SSI）事务。所以，总的来说，PostgreSQL支持三个隔离级别。请注意，不支持“已读未提交”（在一些商业数据库中仍然是默认的）：如果你试图启动一个“已读未提交”的事务，PostgreSQL会默默地映射到“已读提交”。让我们回到可序列化的隔离级别。

如果你想了解更多关于这个隔离级别的信息，可以查看<https://wiki.postgresql.org/wiki/Serializable>。

可序列化隔离背后的想法很简单；如果已知一个事务在只有一个用户时能正常工作，那么在选择这个隔离级别时，它在并发的情况下也会正常工作。然而，用户必须做好准备；事务可能会失败（通过设计）并出错。除此以外，还必须支付性能损失。

只有当你对数据库引擎内部发生的事情有相当的了解时，才考虑使用可序列化的隔离。

5. 观察死锁和类似问题

死锁是一个重要的问题，可能发生在每个数据库中。基本上，如果两个事务必须互相等待，就会发生死锁。

在本节中，你将看到这种情况如何发生。假设我们有一个包含两行的表。

```
CREATE TABLE t_deadlock (id int);
INSERT INTO t_deadlock VALUES (1), (2);
```

以下示例显示了可能发生的情况：

Transaction 1	Transaction 2
BEGIN;	BEGIN;
UPDATE t_deadlock SET id = id * 10 WHERE id = 1;	UPDATE t_deadlock SET id = id * 10 WHERE id = 2;
UPDATE t_deadlock SET id = id * 10 WHERE id = 2;	
Waiting on transaction 2	UPDATE t_deadlock SET id = id * 10 WHERE id = 1;
Waiting on transaction 2	Waiting on transaction 1
	Deadlock will be resolved after 1 second (deadlock_timeout)
COMMIT;	ROLLBACK;

一旦检测到死锁，将显示以下错误消息

```

psql: ERROR: deadlock detected
DETAIL: Process 91521 waits for ShareLock on transaction 903;
blocked by process 77185.
Process 77185 waits for ShareLock on transaction 905;
blocked by process 91521.
HINT: See server log for query details.
CONTEXT: while updating tuple (0,1) in relation "t_deadlock"

```

PostgreSQL甚至好心地告诉我们哪一行引起了冲突。在我的例子中，万恶的根源是一个元组，`(0, 1)`。你在这里看到的是ctid，它是表中某一行的唯一标识符。它告诉我们一个行在表中的物理位置。在这个例子中，它是第一块`(0)`中的第一行。

如果该行对你的事务仍是可见的，甚至有可能查询该行。下面是它的工作原理。

```

test=# SELECT ctid, * FROM t_deadlock WHERE ctid = '(0, 1)';
 ctid | id
-----+-
 (0,1) | 10
 (1 row)

```

请记住，如果某条记录已经被删除或修改，这个查询可能不会返回。

然而，这并不是唯一可能导致死锁的情况，而是可能导致事务失败。事务也可能因为各种原因而不能被序列化。下面的例子显示了可能发生的情况。为了使这个例子有效，我假设你仍然有两行，`id=1`和`id=2`。

Transaction 1	Transaction 2
BEGIN ISOLATION LEVEL REPEATABLE READ;	
SELECT * FROM t_deadlock;	
Two rows will be returned	
	DELETE FROM t_deadlock;
SELECT * FROM t_deadlock;	
Two rows will be returned	
DELETE FROM t_deadlock;	
The transaction will error out	
ROLLBACK; - we cannot COMMIT anymore	

在这个例子中，有两个并发的事务在工作。只要第一个事务只是在选择数据，一切都很好，因为PostgreSQL可以很容易地保持静态数据的假象。但是如果第二个事务提交了一个DELETE命令，会发生什么？只要只有读，就没有问题。当第一个事务试图删除或修改此时已经死亡的数据时，麻烦就开始了。对于PostgreSQL来说，唯一的解决办法是由于我们的事务造成的冲突而出错。

```

test=# DELETE FROM t_deadlock;
psql: ERROR: could not serialize access due to concurrent update

```

实际上，这意味着终端用户必须准备好处理错误的事务。如果出了问题，正确编写的应用程序必须能够再次尝试

6.利用咨询锁

PostgreSQL有高效和复杂的交易机制，能够以真正精细和高效的方式处理锁。几年前，人们想出了用这种代码来使应用程序相互同步的想法。因此，咨询锁就诞生了。

当使用咨询锁时，重要的是要提到，它们不会像普通锁那样在COMMIT时消失。因此，确保解锁是以一种完全可靠的方式正确完成的，这一点真的很重要。

如果你决定使用咨询锁，你真正锁定的是一个数字。所以，这不是关于行或数据，它真的只是一个数字。下面是它的工作原理。

Session 1	Session 2
BEGIN;	
SELECT pg_advisory_lock(15);	SELECT pg_advisory_lock(15);
	It has to wait
COMMIT;	It still has to wait
SELECT pg_advisory_unlock(15);	It is still waiting
	Lock is taken

第一个事务将锁定15。第二个事务必须等待，直到这个数字再次被解锁。第二个事务甚至会在第一个事务提交后等待。这一点非常重要，因为你不能依靠事务的结束会很好地、奇迹般地为你解决事情。如果你想解锁所有被锁定的数字，PostgreSQL提供了pg_advisory_unlock_all()函数来做这件事。

```
test=# SELECT pg_advisory_unlock_all();
pg_advisory_unlock_all
-----
(1 row)
```

有时，你可能想看看你是否能得到一个锁，如果不可能的话就出错。为了达到这个目的，PostgreSQL提供了一些函数；要查看所有这些可用函数的列表，请在命令行输入

```
\df *try*advisory*
```

7.优化存储和管理清理

事务是PostgreSQL系统的一个组成部分。然而，事务是有一个小小的代价的。正如我们在本章中已经表明的，有时，并发用户会得到不同的数据。不是每个人都会得到查询返回的相同数据。除此之外，DELETE和UPDATE不允许实际覆盖数据，因为ROLLBACK不起作用。如果你碰巧处于一个大型的DELETE操作中，你不能确定你是否能够COMMIT。

除此之外，在你执行DELETE操作时，数据仍然是可见的，有时甚至在你的修改早已完成后，数据仍然可见。因此，这意味着清理工作必须以异步方式进行。事务不能清理自己的烂摊子，任何COMMIT/ROLLBACK都可能太早，无法处理死行。

解决这个问题的方法是VACUUM。下面的代码块为你提供了一个语法概述

```
test=# \h VACUUM
Command: VACUUM
Description: garbage-collect and optionally analyze a database
Syntax:
VACUUM [ ( option [, ...] ) ] [ table_and_columns [, ...] ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ ANALYZE ] [ table_and_columns [, ...] ]
where option can be one of:
  FULL [ boolean ]
  FREEZE [ boolean ]
  VERBOSE [ boolean ]
  ANALYZE [ boolean ]
  DISABLE_PAGE_SKIPPING [ boolean ]
  SKIP_LOCKED [ boolean ]
  INDEX_CLEANUP [ boolean ]
  TRUNCATE [ boolean ]
  PARALLEL integer
and table_and_columns is:
  table_name [ ( column_name [, ...] ) ]
URL: https://www.postgresql.org/docs/13/sql-vacuum.html
```

VACUUM将访问所有可能包含修改的页面，并找到所有的死空间。然后，找到的自由空间被关系的自由空间图(FSM)所跟踪。

请注意，在大多数情况下，VACUUM不会缩减表的大小。相反，它将跟踪并找到现有存储文件内的自由空间。

在VACUUM之后，表通常会有相同的大小。如果一个表的末尾没有有效的行，文件大小会下降，尽管这种情况很少。这不是规则，而是例外。

这对终端用户意味着什么，将在本章的“工作中观察VACUUM”小节中概述。

7.1 配置 VACUUM 和 autovacuum

在PostgreSQL项目的早期，人们不得不手动运行VACUUM。幸运的是，那些日子早就过去了。现在，管理员可以依靠一个叫做autovacuum的工具，它是PostgreSQL服务器基础设施的一部分。它自动处理清理工作并在后台工作。它每分钟唤醒一次(见postgresql.conf中autovacuum_naptime = 1)，检查是否有工作要做。如果有工作，autovacuum将最多分叉三个工作进程(见postgresql.conf中的autovacuum_max_workers)。

主要的问题是，autovacuum什么时候触发工作进程的创建？

实际上，autovacuum进程本身并没有分叉进程。相反，它告诉主进程这样做。这样做是为了避免在发生故障时出现僵尸进程，并提高健壮性。

这个问题的答案同样可以在postgresql.conf中找到，如以下代码所示。

```
autovacuum_vacuum_threshold = 50
autovacuum_analyze_threshold = 50
autovacuum_vacuum_scale_factor = 0.2
autovacuum_analyze_scale_factor = 0.1
autovacuum_vacuum_insert_threshold = 1000
```

`autovacuum_vacuum_scale_factor`命令告诉PostgreSQL，如果一个表有20%的数据被改变，就值得进行vacuuming。问题是，如果一个表只有一条记录，一个变化已经是100%了。为了清理一条记录而分出一个完整的过程是完全没有意义的。因此，`autovacuum_vacuum_threshold`说，我们需要20%，而且这个20%必须至少是50行。否则，VACUUM就不会启动。当涉及到创建优化器统计时，也使用了同样的机制。我们需要10%和至少50行来证明新的优化器统计数据的合理性。理想情况下，`autovacuum`会在正常的VACUUM期间创建新的统计信息，以避免不必要的查表次数。

然而，还有更多。在过去，`autovacuum`不会被仅由INSERT语句组成的工作负载所触发，这可能是一个大问题。新增的`autovacuum_vacuum_insert_threshold`参数正是为了解决这种问题。现在，即使数据库中只有INSERT语句，PostgreSQL 13也会触发自动真空活动。

7.2 深入研究与事务环境相关的问题

在`postgresql.conf`中还有两个设置，对于真正利用PostgreSQL来说，理解它们是相当重要的。正如我们已经说过的，理解VACUUM是性能的关键。

```
autovacuum_freeze_max_age = 200000000
autovacuum_multixact_freeze_max_age = 400000000
```

要理解整个问题，重要的是要理解PostgreSQL如何处理并发。PostgreSQL的事务机制是基于对事务ID和事务所处状态的比较。

让我们看一个例子。如果我是事务ID 4711，如果你碰巧是4712，我不会看到你，因为你还在运行。如果我是交易ID 4711，但你是事务ID 3900，我将看到你。如果你的事务失败了，我可以安全地忽略所有由你失败的事务产生的行。

问题如下：事务ID是有限的，不是无限的。在某些时候，它们会开始缠绕在一起。在现实中，这意味着第5个事务可能实际上是在第8亿个事务之后。PostgreSQL如何知道哪个是第一个？它通过存储一个水印来实现。在某些时候，这些水印会被调整，而这正是VACUUM开始发挥作用的时候。通过运行VACUUM（或`autovacuum`），你可以确保水印的调整方式总是有足够的未来交易ID可以使用。

不是每个事务都会增加事务ID计数器。只要一个事务还在读，它就只有一个虚拟事务ID。这确保了事务ID不会被过快烧毁。

`autovacuum_freeze_max_age`命令定义了一个表的`pg_class.relfrozenxid`字段在强制进行VACUUM操作之前可以达到的最大事务数（age），以防止表内的事务ID缠绕。这个值是相当低的，因为它对堵塞的清理也有影响（堵塞或提交日志是一个数据结构，每个事务存储两个比特，这表明一个事务是否正在运行，中止，提交，或仍然在一个子事务中）。

`autovacuum_multixact_freeze_max_age`命令配置了表的`pg_class.relminmxid`字段在强制进行VACUUM操作之前可以达到的最大年龄，以防止表内的multixact ID缠绕。冻结元祖是一个重要的性能问题，在第6章“优化查询以获得良好的性能”中会有更多关于这个过程的内容，我们将讨论查询优化。

一般来说，在保持操作安全性的同时，试图减少VACUUM的负载是一个好主意。对大表进行VACUUM操作可能会很昂贵，因此关注这些设置是非常有意义的。

7.3 关于 VACUUM FULL 的一句话

你也可以用VACUUM FULL来代替普通的VACUUM。然而，我真的想指出，VACUUM FULL实际上锁定了表并重写了整个关系。在一个小表的情况下，这可能不是一个问题。但是，如果你的表很大，表锁可以在几分钟内杀死你。VACUUM FULL会阻止即将到来的写操作，因此，一些与你的数据库交谈的人可能会感觉到它实际上已经停机了。因此，我们建议要非常谨慎。

为了摆脱VACUUM FULL，我推荐你查看`pg_squeeze` (<http://www.cybertec.at/introducing-pg-squeeze-a-postgresql-extension-to-auto-rebuild-bloated-tables/>)，它可以重写一个表而不阻塞写入。

7.4 观察VACUUM的工作情况

现在，是时候看看VACUUM的操作了。我把这部分内容放在这里，因为我作为PostgreSQL顾问和支持者实际工作（<http://www.postgresql-support.com/>）表明，大多数人对存储方面发生的事情只有非常模糊的认识。

再次强调这一点，在大多数情况下，VACUUM不会缩减你的表；空间通常不会返回到文件系统中。

下面是我的例子，它显示了如何用自定义的自动真空设置创建一个小表。该表充满了100000条记录。

```
CREATE TABLE t_test (id int) WITH (autovacuum_enabled = off);
INSERT INTO t_test
SELECT * FROM generate_series(1, 100000);
```

我们的想法是创建一个包含100000行的简单表。注意，可以关闭特定表的自动真空功能。通常情况下，这对大多数应用来说不是一个好主意。然而，在一个角落里，autovacuum_enabled = off是有意义的。只需考虑一个生命周期很短的表。如果开发者已经知道整个表将在几秒钟内被丢弃，那么清理元祖就没有意义了。在数据仓库中，如果你使用表作为暂存区域，就会出现这种情况。在这个例子中，VACUUM被关闭，以确保在后台没有任何事情发生。你所看到的一切是由我触发的，而不是由某个进程触发的。

首先，考虑通过使用以下命令检查表的大小。

```
test=# SELECT pg_size.pretty(pg_relation_size('t_test'));
pg_size.pretty
-----
 3544 kB
(1 row)
```

pg_relation_size命令返回一个表的大小，单位是字节。pg_size.pretty命令将把这个数字转变成人类可读的数字。

然后，表内的所有行将使用一个简单的UPDATE语句进行更新，如以下代码所示。

```
test=# UPDATE t_test SET id = id + 1;
UPDATE 100000
```

所发生的事情对理解PostgreSQL非常重要。数据库引擎必须复制所有的行。为什么呢？首先，我们不知道事务是否会成功，所以数据不能被覆盖。第二个重要的方面是，一个并发的事务可能还在看到旧版本的数据。UPDATE操作会复制行。从逻辑上讲，改变之后，表的大小会变大。

```
test=# SELECT pg_size.pretty(pg_relation_size('t_test'));
pg_size.pretty
-----
 7080 kB
(1 row)
```

在UPDATE之后，人们可能会尝试将空间返回到文件系统。

```
test=# VACUUM t_test;
VACUUM
```

正如我们前面所说，在大多数情况下，VACUUM不会将空间返回到文件系统。相反，它将允许空间被重新使用。因此，该表根本不会缩减。

```
test=# SELECT pg_size.pretty(pg_relation_size('t_test'));
pg_size.pretty
-----
7080 kB
(1 row)
```

然而，下一个UPDATE不会使表增长，因为它将吃掉表内的自由空间。只有第二次UPDATE才会使表再次增长，因为所有的空间都没有了，所以需要额外的存储。

```
test=# UPDATE t_test SET id = id + 1;
UPDATE 100000
test=# SELECT pg_size.pretty(pg_relation_size('t_test'));
pg_size.pretty
-----
7080 kB
(1 row)
test=# UPDATE t_test SET id = id + 1;
UPDATE 100000
test=# SELECT pg_size.pretty(pg_relation_size('t_test'));
pg_size.pretty
-----
10 MB
(1 row)
```

如果我必须决定你在读完这本书后应该记住一件事，那就是这一点。了解存储是性能和一般管理的关键。

让我们再运行一些查询。

```
VACUUM t_test;
UPDATE t_test SET id = id + 1;
VACUUM t_test;
```

同样，尺寸也没有变化。让我们看看表格里面有什么。

```
test=# SELECT ctid, * FROM t_test ORDER BY ctid DESC;
ctid | id
-----+-----
...
(1327, 46) | 112
(1327, 45) | 111
(1327, 44) | 110
...
(884, 20) | 99798
(884, 19) | 99797
...
```

ctid命令是一个行在磁盘上的物理位置。通过使用ORDER BY ctid DESC，你基本上会按照物理顺序向后读取表。你为什么要关心这个问题呢？因为在表的末端有一些非常小的值和一些非常大的值。下面的代码显示了当数据被删除时，表的大小是如何变化的

```
test=# DELETE FROM t_test
 WHERE id > 99000
 OR id < 1000;
DELETE 1999
test=# VACUUM t_test;
VACUUM
test=# SELECT pg_size_pretty(pg_relation_size('t_test'));
 pg_size_pretty
-----
 3504 kB
(1 row)
```

虽然只有2%的数据被删除，但表的大小却减少了三分之二。其原因是，如果VACUUM只发现表中某个位置之后的死行，它可以将空间返回到文件系统中。这是唯一的一种情况，在这种情况下，你会真正看到表的大小下降。当然，普通用户无法控制数据在磁盘上的物理位置。因此，除非所有的行都被删除，否则存储消耗很可能会保持一定程度的不变。

到底为什么在表的最后会有这么多的小值和大值？在表最初填充了100,000行后，最后一个区块并没有完全填满，所以第一个UPDATE会把最后一个区块填满变化。这就把表的末尾洗了一下。在这个精心制作的例子中，这就是表末尾奇怪布局的原因。

在现实世界的应用中，这一观点的影响怎么强调都不为过。不真正了解存储，就没有性能调整。

7.5 通过使用老旧的快照来限制事务

VACUUM做得很好，它将根据需要回收自由空间。然而，VACUUM什么时候才能真正清理出行并将其变成自由空间呢？规则是这样的：如果一个行不能再被任何人看到，它就可以被回收。在现实中，这意味着所有不再被看到的东西，即使是最古老的活动事务，都可以被认为是真正的死亡。

这也意味着，真正长的事务可以推迟相当长的时间来清理。逻辑上的后果是表的膨胀。表的增长将超出比例，性能将趋于下降。幸运的是，从PostgreSQL 9.6开始，数据库有一个很好的功能，允许管理员智能地限制一个事务的持续时间。Oracle管理员将熟悉快照太旧的错误。从PostgreSQL 9.6开始，这个错误信息也有了。然而，它更像是一个功能，而不是不良配置的意外副作用（在Oracle中它实际上是）。

为了限制快照的寿命，你可以利用PostgreSQL的配置文件postgresql.conf中的一个设置，其中有所有需要的配置参数。

```
old_snapshot_threshold = -1
# 1min-60d; -1 disables; 0 is immediate
```

如果这个变量被设置了，事务将在一定时间后失败。请注意，这个设置是在实例层面上的，它不能在会话中设置。通过限制事务的年龄，疯狂的长事务的风险将大大降低。

7.6 使用更多的 VACUUM 特性

多年来，VACUUM一直在稳步改进。在本节中，你将了解到一些最近的改进。

在许多情况下，VACUUM可以跳过页面。当可见性地图显示一个区块对所有人都是可见的时候，这一点尤其真实。VACUUM也可能跳过一个被其他事务大量使用的页面。DISABLE_PAGE_SKIPPING禁止这种行为，并确保所有的页面在这次运行中被清理。

还有一种改进VACUUM的方法是使用SKIP_LOCKED：这里的想法是确保VACUUM不损害并发性。如果使用SKIP_LOCKED，VACUUM将自动跳过不能立即锁定的关系，从而避免冲突解决。这种功能在严重并发的情况下可能非常有用。

VACUUM的一个重要而有时被忽视的方面是需要清理索引。在VACUUM成功地处理了一个堆之后，索引被处理了。如果你想防止这种情况发生，你可以利用INDEX_CLEANUP。默认情况下，INDEX_CLEANUP是真的，但是根据你的工作负载，你可能会决定在一些罕见的情况下跳过索引清理。那么，那些罕见的情况是什么呢？为什么有人可能不想清理索引呢？答案很简单：如果你的数据库有可能很快因为事务缠绕而关闭，那么尽快运行VACUUM是有意义的。如果你在停机和某种推迟的清理之间有一个选择，你应该选择快速的VACUUM来保持你的数据库的活力。

8.总结

在本章中，你了解了事务、锁定及其逻辑含义，以及PostgreSQL事务机制在存储、并发和管理方面的一般架构。你看到了行是如何被锁定的，以及PostgreSQL中的一些功能。

在第3章 "使用索引" 中，你将了解到数据库工作中最重要的主题之一：索引。你还将了解PostgreSQL的查询优化器，以及各种类型的索引和它们的行为。

9.问题

- 事务的目的是什么？
- 在PostgreSQL中一个事务可以有多长？
- 什么是事务隔离？
- 我们应该避免表锁吗？
- 事务与VACUUM有什么关系？

这些问题的答案可以在GitHub仓库中找到 (<https://github.com/PacktPublishing/Mastering-PostgreSQL-13-Fourth-Edition>) 。

使用索引

1 了解简单的查询和成本模型

- 1.1 使用 EXPLAIN
- 1.2 深入了解 PostgreSQL 成本模型
- 1.3 部署简单索引
- 1.4 使用排序输出
- 1.5 一次使用多个索引
- 1.6 有效地使用位图扫描
- 1.7 以智能方式使用索引
- 1.8 了解索引去重

2 使用聚类表提高速度

- 2.1 聚类表
- 2.2 使用仅索引扫描

3 了解额外的B-树功能

- 3.1 组合索引
- 3.2 添加功能索引
- 3.3 减少空间消耗
- 3.4 索引时添加数据

4 介绍运算符类

- 4.1 为 B 树创建运算符类
- 4.2 创建新的运算符
- 4.3 创建运算符类
- 4.4 测试自定义运算符类

5 了解PostgreSQL索引类型

- 5.1 哈希索引
- 5.2 GiST 索引
 - 5.2.1 了解 GiST 的工作原理
 - 5.2.2 扩展 GiST
- 5.3 GIN 索引
 - 5.3.1 扩展 GIN
- 5.4 SP-GiST 索引
- 5.5 BRIN索引
 - 5.5.1 扩展 BRIN 索引
- 5.6 添加额外的索引

6 用模糊搜索实现更好的答案

- 6.1 利用 pg_trgm
- 6.2 加快 LIKE 查询
- 6.3 处理正则表达式

7 了解全文搜索

- 7.1 比较字符串
- 7.2 定义 GIN 索引
- 7.3 调试搜索
- 7.4 收集单词统计信息
- 7.5 利用排除运算符

8 总结

9 问题

在第2章 "理解事务和锁定"中，你了解了并发性和锁定。在这一章中，是时候直面索引了。这个话题的重要性怎么强调都不为过--索引是（而且很可能一直是）每个数据库工程师生活中最重要的话题之一。

经过20多年专业的、全职的PostgreSQL咨询和PostgreSQL 24/7支持经验 (www.cybertec-postgresql.com)，我可以肯定地说一件事--坏的索引是性能不好的主要来源。当然，调整内存参数和所有这些是很重要的。但是，如果索引使用不当，这一切都是徒劳的。缺少索引是根本无法替代的。为了明确我的观点：如果没有适当的索引，就没有办法实现良好的性能，所以如果性能不好，一定要检查索引。

这就是专门用一整章来讨论索引的背后原因。这将给你带来尽可能多的启示。

在这一章中，我们将涵盖以下主题。

- 了解简单的查询和成本模型
- 使用聚类表提高速度
- 了解额外的B-树功能
- 介绍运算符类
- 了解PostgreSQL索引类型
- 用模糊搜索实现更好的答案
- 了解全文搜索

在本章结束时，你将了解在PostgreSQL中如何有益地使用索引。

1 了解简单的查询和成本模型

在本节中，我们将开始使用索引。为了了解事情是如何运作的，需要一些测试数据。下面的代码片断显示了如何轻松创建数据。

```
test=# DROP TABLE IF EXISTS t_test;
DROP TABLE
test=# CREATE TABLE t_test (id serial, name text);
CREATE TABLE
test=# INSERT INTO t_test (name) SELECT 'hans'
   FROM generate_series(1, 2000000);
INSERT 0 2000000
test=# INSERT INTO t_test (name) SELECT 'paul'
   FROM generate_series(1, 2000000);
INSERT 0 2000000
```

在第一行，创建了一个简单的表。使用了两列；第一列是一个自动递增列，只是不断地创建数字，第二列是一个将用静态值填充的列。

generate_series函数将生成从1到200万的数字。因此，在这个例子中，hans的200万个静态值和paul的200万个静态值被创建了

总共添加了400万行：

```
test=# SELECT name, count(*) FROM t_test GROUP BY 1;
 name | count
-----+-----
 hans | 2000000
 paul | 2000000
(2 rows)
```

这400万行有一些很好的属性，我们将在本章中使用这些属性。ID是按升序排列的数字，而且只有两个不同的名字。

让我们运行一个简单的查询。

```
test=# \timing
Timing is on.
test=# SELECT * FROM t_test WHERE id = 432332;
 id | name
-----
 432332 | hans
(1 row)
Time: 176.949 ms
```

在这种情况下，计时命令将告诉 psql 显示查询的运行时间。

这不是服务器上的真实执行时间，而是psql测量的时间。如果是非常短的查询，网络延迟可能是总时间的很大一部分，所以必须考虑到这一点。

1.1 使用 EXPLAIN

在这个例子中，读取400万行已经花费了100多毫秒的时间。从性能的角度来看，这完全是一场灾难。为了弄清楚出了什么问题，PostgreSQL提供了EXPLAIN命令，其定义如下代码所示。

```
test=# \h EXPLAIN
Command: EXPLAIN
Description: show the execution plan of a statement
Syntax:
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
where option can be one of:
ANALYZE [ boolean ]
VERBOSE [ boolean ]
COSTS [ boolean ]
SETTINGS [ boolean ]
BUFFERS [ boolean ]
WAL [ boolean ]
TIMING [ boolean ]
SUMMARY [ boolean ]
FORMAT { TEXT | XML | JSON | YAML }
URL: https://www.postgresql.org/docs/13/sql-explain.html
```

当你感觉到一个查询的性能不好时，EXPLAIN将帮助你揭示性能问题的真正原因。

下面是它的工作原理。

```
test=# EXPLAIN SELECT * FROM t_test WHERE id = 432332;
 QUERY PLAN
-----
Gather (cost=1000.00..43455.43 rows=1 width=9)
Workers Planned: 2
-> Parallel Seq Scan on t_test (cost=0.00..42455.33 rows=1 width=9)
  Filter: (id = 432332)
(4 rows)
```

你在这个列表中看到的是一个执行计划。在PostgreSQL中，一条SQL语句将分四个阶段执行。以下组件在工作。

- 解析器将检查语法错误和明显的问题。
- 重写系统负责处理规则（视图和其他东西）。

- 优化器将找出如何以最有效的方式执行查询，并制定一个计划。
- 优化器提供的计划将被执行器用来最终创建结果。

EXPLAIN的目的是看规划器想出了什么办法来有效地运行查询。在我的例子中，PostgreSQL将使用一个并行的顺序扫描。这意味着两个工作者将合作，一起进行过滤条件的工作。然后，部分结果会通过一个叫做聚集节点的东西联合起来，这是在PostgreSQL 9.6中引入的（它是并行查询基础设施的一部分）。如果你更仔细地看这个计划，你会看到PostgreSQL在计划的每个阶段期望有多少行（在这个例子中，行=1；也就是说，将返回一条行）。

在PostgreSQL 9.6到10.0中，并行工作者的数量将由表的大小决定。操作越大，PostgreSQL将启动更多的并行工作者。对于一个非常小的表，不使用并行操作，因为它将产生太多的开销。

平行性不是必须的。通过将以下变量设置为0，总是可以减少并行工作者的数量以模仿PostgreSQL 9.6之前的行为。

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

请注意，这种改变没有副作用，因为它只在你的会话中发生。当然，你也可以在postgresql.conf文件中做这个改动，但我建议不要这样做，因为你可能会失去很多由并行查询提供的性能。

1.2 深入了解 PostgreSQL 成本模型

如果只使用一个CPU，执行计划会是这样的。

```
test=# EXPLAIN SELECT * FROM t_test WHERE id = 432332;
QUERY PLAN
-----
Seq Scan on t_test (cost=0.00..71622.00 rows=1 width=9)
  Filter: (id = 432332)
  (2 rows)
```

PostgreSQL将按顺序读取（顺序扫描）整个表并应用过滤器。它预计该操作将花费71622个罚分。现在，这意味着什么呢？罚分（或成本）主要是一个抽象的概念。它们需要用来比较执行查询的不同方法。如果一个查询可以被执行器以许多不同的方式执行，PostgreSQL将通过承诺最低的成本来决定执行计划。现在的问题是，PostgreSQL是如何得到71622分的？

下面是它的工作原理。

```
test=# SELECT pg_relation_size('t_test') / 8192.0;
?column?
-----
21622.000000
(1 row)
```

pg_relation_size函数将返回表的大小，单位是字节。鉴于这个例子，你可以看到这个关系由21622个块组成（每个块8000个）。根据成本模型，PostgreSQL将为每一个必须按顺序读取的块增加一个成本。

影响这一点的配置参数如下。

```
test=# SHOW seq_page_cost;
seq_page_cost
-----
1
(1 row)
```

然而，从磁盘上读取几个块并不是我们要做的全部。还需要应用过滤器，并通过CPU发送这些行。下面的代码块中显示的两个参数说明了这些成本。

```
test=# SHOW cpu_tuple_cost;
cpu_tuple_cost
-----
0.01
(1 row)

test=# SHOW cpu_operator_cost;
cpu_operator_cost
-----
0.0025
(1 row)
```

这导致以下计算：

```
test=# SELECT 21622*1 + 4000000*0.01 + 4000000*0.0025;
?column?
-----
71622.0000
(1 row)
```

正如你所看到的，这正是计划中所显示的数字。成本将由CPU部分和I/O部分组成，它们都将被转化为一个数字。这里重要的是，成本与实际执行无关，所以不可能将成本转化为毫秒。计划员得出的数字其实只是一个估计值。

当然，在这个简短的例子中还概述了一些参数。PostgreSQL对与索引有关的操作也有特殊的参数，如下所示

- `random_page_cost = 4`: 如果PostgreSQL使用了一个索引，通常会涉及大量的随机I/O。在传统的旋转磁盘上，随机读取要比顺序读取重要得多，所以PostgreSQL会相应地考虑它们。注意，在SSD上，随机读和顺序读之间的差别已经不存在了，所以在`postgresql.conf`文件中设置`random_page_cost = 1`可能是有意义的。
- `cpu_index_tuple_cost = 0.005`。如果使用了索引，PostgreSQL也会认为有一些CPU成本的开销

如果你正在利用并行查询，则有更多的成本参数。

- `parallel_tuple_cost = 0.1`: 这定义了将一个元组从一个并行工作进程转移到另一个进程的成本。它基本上说明了在基础设施内移动行的开销。
- `parallel_setup_cost = 1000.0`: 这调整了启动一个工作进程的成本。当然，启动进程来并行运行查询不是免费的，所以这个参数试图模拟那些与进程管理相关的成本。
- `min_parallel_tables_scan_size = 8 MB`: 这定义了一个被考虑用于并行查询的表的最小尺寸。一个表增长得越大，PostgreSQL将使用更多的CPU。表的大小必须是三倍，以允许多一个工作进程。
- `min_parallel_index_scan_size = 512kB`: 这定义了一个索引的大小，这是考虑并行扫描所必须的。

1.3 部署简单索引

发动更多的工作进程来扫描更大的表，有时并不是解决办法。读取整个表来寻找单行通常不是一个好主意。

因此，创建索引是有意义的。

```
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
test=# SELECT * FROM t_test WHERE id = 43242;
 id | name
-----
 43242 | hans
(1 row)
Time: 0.259 ms
```

PostgreSQL使用Lehman-Yao的高并发B树来做标准索引 (<https://www.cs.uoc.gr/~hy460/pdf/p650-lehman.pdf>)。连同一些PostgreSQL特有的优化，这些树为终端用户提供了出色的性能。最重要的是，Lehman-Yao允许你同时在同一个索引上运行许多操作（读和写），这有助于极大地提高吞吐量。

然而，索引并不是免费的。

```
test=# \di+
List of relations
 Schema | Name | Type | Owner | Table | Persistence | Size | Description
-----+-----+-----+-----+-----+-----+-----+-----+
 public | idx_id | index | hs | t_test | permanent | 86 MB |
(1 row)
```

正如你所看到的，我们的包含400万行的索引将占用86MB的磁盘空间。除此以外，对表的写入也会变慢，因为索引必须一直保持同步。

换句话说，如果你插入到一个具有20个索引的表中，你还必须记住，我们必须在INSERT时向所有这些索引写入，这就严重降低了写入速度

随着版本11的引入，PostgreSQL现在支持并行索引创建。它可以利用一个以上的CPU核心来建立索引，从而大大加快了这个过程。目前，只有当你想建立一个普通的B树时才有可能，还没有对其他索引类型的支持。然而，这在将来很可能会改变。用于控制并行程度的参数是max_parallel_maintenance_workers。它告诉PostgreSQL它可以使用多少个进程作为一个上限。

1.4 使用排序输出

B树索引不仅用于查找行；它们还用于将排序后的数据提供给流程的下一阶段：

```
test=# EXPLAIN SELECT *
  FROM t_test
 ORDER BY id DESC
 LIMIT 10;
QUERY PLAN
-----
Limit (cost=0.43..0.74 rows=10 width=9)
 -> Index Scan Backward using idx_id on t_test
(cost=0.43..125505.43 rows=4000000 width=9)
(2 rows)
```

在这种情况下，索引已经按照正确的排序顺序返回数据，因此没有必要对整个数据集进行排序。读取索引的最后10行就足以回答这个查询。实际上，这意味着有可能在几分之一秒的时间内找到一个表的前N行。

然而，ORDER BY并不是唯一需要排序输出的操作。min和max函数也都是关于排序输出的，所以索引也可以用来加快这两个操作的速度。下面是一个例子

```
test=# explain SELECT min(id), max(id) FROM t_test;
QUERY PLAN

Result (cost=0.92..0.93 rows=1 width=8)
InitPlan 1 (returns $0)
-> Limit (cost=0.43..0.46 rows=1 width=4)
-> Index Only Scan using idx_id on t_test
(cost=0.43..113880.43 rows=4000000 width=4)
Index Cond: (id IS NOT NULL)
InitPlan 2 (returns $1)
-> Limit (cost=0.43..0.46 rows=1 width=4)
-> Index Only Scan Backward using idx_id on t_test t_test_1
(cost=0.43..113880.43 rows=4000000 width=4)
Index Cond: (id IS NOT NULL)
(9 rows)
```

在PostgreSQL中，一个索引（更准确的说是一个B-树）可以按正常顺序或向后读取。现在的问题是，B树可以被看作是一个排序的列表。所以，很自然地，最低值在开始，最高值在结束。因此，min和max是加速的最佳选择。同样值得注意的是，在这种情况下，主表根本不需要被引用。

在SQL中，许多操作都依赖于排序的输入；因此，理解这些操作是至关重要的，因为在索引方面有严重的影响。

1.5 一次使用多个索引

到现在为止，你已经看到一次使用一个索引了。然而，在许多现实世界的情况下，这远远不够。有一些情况需要在数据库中使用更多的逻辑。

PostgreSQL允许在一个查询中使用多个索引。当然，如果许多列同时被查询，这是有意义的。然而，情况并不总是这样的。也可能发生这样的情况：一个索引被多次使用来处理同一个列。

下面是一个例子。

```
test=# explain SELECT * FROM t_test WHERE id = 30 OR id = 50;
QUERY PLAN

Bitmap Heap Scan on t_test (cost=8.88..16.85 rows=2 width=9)
Recheck Cond: ((id = 30) OR (id = 50))
-> BitmapOr (cost=8.88..8.88 rows=2 width=0)
-> Bitmap Index Scan on idx_idv
(cost=0.00..4.44 rows=1 width=0)
Index Cond: (id = 30)
-> Bitmap Index Scan on idx_id (cost=0.00..4.44 rows=1 width=0)
Index Cond: (id = 50)
(7 rows)
```

这里的重点是，id列需要两次。首先，该查询寻找30，然后寻找50。正如你所看到的，PostgreSQL会去进行位图扫描。

位图扫描和位图索引不一样，有良好的Oracle背景的人可能知道。它们是两种完全不同的东西，没有任何共同之处。位图索引是Oracle的一种索引类型，而位图扫描是一种扫描方法。

位图扫描背后的想法是，PostgreSQL将扫描第一个索引，收集一个包含数据的块（表的页面）列表。然后，下一个索引将被扫描，再次编制一个块的列表。这是为所需的多个索引所做的。在OR的情况下，这些列表将被统一，给我们留下一个包含数据的长块列表。使用这个列表，表将被扫描以检索这些块。

现在的麻烦是，PostgreSQL已经检索到了比需要的多得多的数据。在我们的例子中，查询将寻找两行；然而，几个块可能已经被位图扫描返回。因此，执行器将进行重新检查以过滤掉这些行，也就是不满足我们条件的行。

位图扫描也将对AND条件或AND和OR的混合条件起作用。然而，如果PostgreSQL看到一个AND条件，它不一定强迫自己进行位图扫描。让我们假设我们有一个查询，寻找住在奥地利的所有人和一个有特定ID的人。在这里使用两个索引真的没有意义，因为在搜索完ID之后，剩下的数据真的不多。扫描两个索引的成本会更高，因为有800万人（包括我）住在奥地利，从性能的角度来看，读取这么多行来寻找一个人是非常没有意义的。好消息是，PostgreSQL优化器会通过比较不同选项和潜在索引的成本来为你做出所有这些决定，所以不需要担心

1.6 有效地使用位图扫描

现在自然产生的问题是，位图扫描何时最有利，何时被优化器选择？从我的观点来看，实际上只有两种使用情况。

- 为了避免重复获取相同的区块
- 为了结合相对较差的条件

第一种情况是很常见的。假设你要找的是所有讲某种语言的人。在这个例子中，我们可以假设存储在索引中的所有人中，有10%的人讲所需的语言。扫描索引将意味着必须重新扫描表中的一个块，因为许多熟练的发言者可能被存储在同一个块中。通过应用位图扫描，可以确保一个特定的块只被使用一次，这当然会带来更好的性能。

第二个常见的用例是一起使用相对较弱的标准。假设我们要找的是20到30岁之间拥有黄色衬衫的所有人。现在，也许15%的人在20到30岁之间，也许15%的人真的拥有一件黄衬衫。按顺序扫描一个表是很昂贵的，所以PostgreSQL可能决定选择两个索引，因为最终的结果可能只包括1%的数据。扫描两个索引可能比读取所有的数据更便宜。在PostgreSQL 10.0中，支持并行位图堆扫描。通常情况下，位图扫描是由相对昂贵的查询使用的。因此，在这个领域增加的并行性是一个巨大的进步，绝对是有益的。

1.7 以智能方式使用索引

到目前为止，应用索引感觉就像圣杯，总是神奇地提高性能。然而，事实并非如此。在某些情况下，索引也可能是相当无意义的。

在更深入地挖掘事情之前，这里是我们在这个例子中使用的数据结构。记住，只有两个不同的名字和唯一的ID。

```
test=# \d t_test
Table "public.t_test"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
id   | integer | not null | nextval('t_test_id_seq')::regclass
name | text |
Indexes:
"idx_id" btree (id)
```

在这一点上，已经定义了一个索引，它涵盖了id列。在下一步，将对name列进行查询。在这样做之前，将创建一个关于名字的索引。

```
test=# CREATE INDEX idx_name ON t_test (name);
CREATE INDEX
```

现在，是时候看看索引是否被正确使用了；考虑以下代码块。

```
test=# EXPLAIN SELECT * FROM t_test WHERE name = 'hans2';
QUERY PLAN
-----
Index Scan using idx_name on t_test
(cost=0.43..4.45 rows=1 width=9)
Index Cond: (name = 'hans2'::text)
(2 rows)
```

正如预期的那样，PostgreSQL将决定使用该索引。大多数用户都会想到这一点。但是请注意，我的查询说的是hans2。记住，hans2并不存在于表中，查询计划完美地反映了这一点。rows=1表示计划者只期望查询返回一个非常小的数据子集。

表中没有任何一行，但PostgreSQL永远不会估计零行，因为这将使后续的估计变得更加困难，因为计划中其他节点的有用成本计算将接近于不可能。

让我们看看如果我们寻找更多数据会发生什么：

```
test=# EXPLAIN SELECT *
  FROM t_test
 WHERE name = 'hans'
   OR name = 'paul';
QUERY PLAN
-----
Seq Scan on t_test (cost=0.00..81622.00 rows=3000005 width=9)
  Filter: ((name = 'hans'::text) OR (name = 'paul'::text))
(2 rows)
```

在这种情况下，PostgreSQL会直接进行顺序扫描。这是为什么呢？为什么系统忽略了所有的索引？原因很简单：hans和paul构成了整个数据集，因为没有其他的值（PostgreSQL通过检查系统的统计数据知道这一点）。因此，PostgreSQL认为，无论如何都要读取整个表。如果读表就足够了，就没有理由去读所有的索引和全表。

换句话说，PostgreSQL不会因为有一个索引就使用它。PostgreSQL会在有意义的时候使用索引。如果行数较少，PostgreSQL将再次考虑位图扫描和普通索引扫描。

```
test=# EXPLAIN SELECT *
  FROM t_test
 WHERE name = 'hans2'
   OR name = 'paul2';
QUERY PLAN
-----
Bitmap Heap Scan on t_test (cost=8.88..12.89 rows=1 width=9)
  Recheck Cond: ((name = 'hans2'::text) OR (name = 'paul2'::text))
    -> BitmapOr (cost=8.88..8.88 rows=1 width=0)
      -> Bitmap Index Scan on idx_name
```

```
(cost=0.00..4.44 rows=1 width=0)
Index Cond: (name = 'hans2'::text)
-> Bitmap Index Scan on idx_name
(cost=0.00..4.44 rows=1 width=0)
Index Cond: (name = 'paul2'::text)
```

这里要学习的最重要的一点是，执行计划取决于输入值。

它们不是静态的，也不是独立于表内的数据。这是一个非常重要的观点，必须时刻牢记。在现实世界的例子中，计划改变的事实往往是不可预测的运行时间的原因。

1.8 了解索引去重

在PostgreSQL 13及以后的版本中，需要提到的是，并非所有的索引都是平等的。随着第13版的推出，PostgreSQL现在能够对索引条目进行去重。换句话说，一个存储许多相同值的索引将比一个“正常”索引小得多。

```
test=# \di+
List of relations
 Schema | Name | Type | Owner | Table | Persistence | Size | Description
-----+-----+-----+-----+-----+-----+-----+
 public | idx_id | index | hs | t_test | permanent | 86 MB |
 public | idx_name | index | hs | t_test | permanent | 26 MB |
```

较小的索引将通过确保较高的缓存命中率和较好的访问时间来大大改善效率。

2 使用聚类表提高速度

在本节中，你将了解到相关性和聚类表的力量。这是怎么一回事？想象一下，你想读取整个数据区域。这可能是一个特定的时间范围，一个区块，一些IDs，等等。

这些查询的运行时间会有所不同，取决于数据量和磁盘上数据的物理排列。因此，即使你运行的查询返回相同数量的行，两个系统也可能不会在相同的时间范围内提供答案，因为物理磁盘布局可能会产生差异。

下面是一个例子。

```
test=# EXPLAIN (analyze true, buffers true, timing true)
SELECT *
FROM t_test
WHERE id < 10000;
QUERY PLAN
-----
Index Scan using idx_id on t_test (cost=0.43..322.63 rows=9383 width=9)
(actual time=0.469..4.476 rows=9999 loops=1)
Index Cond: (id < 10000)
Buffers: shared hit=3 read=82
Planning Time: 4.450 ms
```

```
Buffers: shared hit=3 read=1
Execution Time: 5.607 ms
(6 rows)
```

正如你所看到的，数据已经以一种有组织和有顺序的方式被加载。数据被添加到一个又一个的ID中，因此可以预计，数据将以一个顺序出现在磁盘上。如果数据被加载到一个使用自动递增列的空表中，这一点是正确的。

你已经看到了EXPLAIN的作用。在这个例子中，EXPLAIN(analyze true, buffers true, and timing true)已经被利用了。这个想法是，analyze不只是显示计划，而且还执行查询，并显示发生了什么。

EXPLAIN analyze对于比较计划员的估计和真正发生的事情是非常完美的。它是弄清计划者是否正确或有偏差的最好方法。缓冲区的真实参数将告诉我们8,000个块中有多少被查询所触及。在这个例子中，总共有85个块被触及。一个共享命中意味着数据来自PostgreSQL的I/O缓存（共享缓冲区）。总的来说，PostgreSQL花了大约5毫秒的时间来检索这些数据。如果你的表中的数据有些随机，会发生什么？事情会改变吗？

要创建一个包含相同数据但顺序随机的表，你可以简单地使用ORDER BY random()。它将确保数据确实在磁盘上被洗过。

```
test=# CREATE TABLE t_random AS SELECT * FROM t_test ORDER BY random();
SELECT 4000000
```

为了确保公平比较，对同一列进行索引：

```
test=# CREATE INDEX idx_random ON t_random (id);
CREATE INDEX
```

为了正常运行，PostgreSQL将需要优化器的统计数据。这些统计数据将告诉PostgreSQL有多少数据，数值如何分布，以及数据在磁盘上是否有关联。为了进一步加快工作速度，我增加了一个VACUUM调用。请注意，VACUUM将在本书后面更深入地讨论

```
test=# VACUUM ANALYZE t_random;
VACUUM
```

现在，让我们运行之前运行的相同查询：

```
test=# EXPLAIN (analyze true, buffers true, timing true)
SELECT * FROM t_random WHERE id < 10000;
QUERY PLAN
-----
Bitmap Heap Scan on t_random (cost=189.13..17782.28 rows=9897 width=9)
(actual time=2.867..103.343 rows=9999 loops=1)
Recheck Cond: (id < 10000)
Heap Blocks: exact=8042
Buffers: shared hit=827 read=7245
-> Bitmap Index Scan on idx_random (cost=0.00..186.66 rows=9897 width=0)
(actual time=1.677..1.677 rows=9999 loops=1)
Index Cond: (id < 10000)
Buffers: shared hit=3 read=27
Planning Time: 0.883 ms
```

```

Buffers: shared hit=10 read=3
Execution Time: 104.331 ms
(10 rows)
test=# EXPLAIN (analyze true, buffers true, timing true)
SELECT * FROM t_random WHERE id < 10000;
QUERY PLAN

Bitmap Heap Scan on t_random (cost=189.13..17782.28 rows=9897 width=9)
(actual time=2.182..8.774 rows=9999 loops=1)
  Recheck Cond: (id < 10000)
  Heap Blocks: exact=8042
  Buffers: shared hit=8072
-> Bitmap Index Scan on idx_random (cost=0.00..186.66 rows=9897 width=0)
(actual time=0.973..0.974 rows=9999 loops=1)
  Index Cond: (id < 10000)
  Buffers: shared hit=30
Planning Time: 0.078 ms
Buffers: shared hit=4
Execution Time: 9.462 ms
(10 rows)

```

我已经执行了两次相同的查询，以显示缓存的影响，但让我们一个接一个地进行如下操作。

这里有几件事情需要观察。首先，在第一次执行过程中，总共需要8042个块，在未缓存的情况下，运行时间已经飙升到超过104毫秒，在缓存的情况下，运行时间为9.462毫秒（共享命中）。正如你所看到的，这里唯一能在一定程度上挽救性能的是，在第二次执行过程中，数据是从内存而不是从磁盘提供的。未缓存的情况是完全I/O绑定的。

然而，还有更多可以看到的。你甚至可以看到计划已经改变了。PostgreSQL现在使用位图扫描而不是普通的索引扫描。这样做是为了减少查询中需要的块的数量，以防止更糟糕的行为。

计划员如何知道数据是如何存储在磁盘上的？`pg_stats`是一个系统视图，包含所有关于列的内容的统计数据。下面的查询显示了相关内容。

```

test=# SELECT tablename, attname, correlation
  FROM pg_stats
 WHERE tablename IN ('t_test', 't_random')
 ORDER BY 1, 2;
 tablename | attname | correlation
-----+-----+
t_random | id | -0.005975342
t_random | name | 0.49059877
t_test | id | 1
t_test | name | 1
(4 rows)

```

你可以看到PostgreSQL照顾到了每一列。视图的内容是由ANALYZE创建的，定义如图，对性能至关重要。

```

test=# \h ANALYZE
Command: ANALYZE
Description: collect statistics about a database
Syntax:
ANALYZE [ ( option [, ...] ) ] [ table_and_columns [, ...] ]

```

```
ANALYZE [ VERBOSE ] [ table_and_columns [, ...] ]
where option can be one of:
  VERBOSE [ boolean ]
  SKIP_LOCKED [ boolean ]
and table_and_columns is:
  table_name [ ( column_name [, ...] ) ]
URL: https://www.postgresql.org/docs/13/sql-analyze.html
```

通常情况下，ANALYZE是在后台使用autovacuum守护程序自动执行的，本书后面将介绍这个问题。回到我们的查询。正如你所看到的，两个表都有两列（id和name）。在t_test.id的情况下，相关性是1，这意味着下一个值在某种程度上取决于前一个。在我的例子中，这些数字只是升序。这同样适用于t_test.name。首先，我们有包含hans的条目，然后是包含paul的条目。因此，所有相同的名字都被存储在一起。

在t_random中，情况完全不同；负相关意味着数据被洗牌了。你也可以看到，名字列的相关度大约是0.5。在现实中，这意味着在表中通常没有相同的名字的直线序列，而是在按物理顺序读取表时，名字一直在切换。

为什么这会导致这么多块被查询击中？答案是比较简单的。如果我们需要的数据不是紧紧地挤在一起，而是均匀地分布在表中，就需要更多的块来提取相同数量的信息，这反过来又导致了更差的性能。

2.1 聚类表

在PostgreSQL中，有一个叫做CLUSTER的命令，它允许我们按照所需的顺序重写一个表。它可以指向一个索引，并以与索引相同的顺序存储数据。

```
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命令已经存在了很多年，并且很好地实现了它的目的。然而，在生产系统上盲目地运行它之前，有一些事情需要考虑。

- CLUSTER命令在运行中会锁定表。当CLUSTER运行时，你不能插入或修改数据。这在生产系统中可能是不可接受的。
- 数据只能根据一个索引来组织。你不能同时按照邮政编码、姓名、身份证件、生日等来排列一个表。这意味着如果有大部分时间都在使用的搜索标准，CLUSTER就有意义。
- 请记住，本书所述的例子更多的是一种最坏的情况。在现实中，集群表和非集群表之间的性能差异将取决于工作负载、检索的数据量、缓存命中率，以及更多的因素。
- 在正常的操作过程中，表的集群状态将不会被保持，因为在正常操作过程中对表进行了改变。相关性通常会随着时间的推移而恶化

以下是如何运行CLUSTER命令的示例：

```
test=# CLUSTER t_random USING idx_random;
CLUSTER
```

根据桌子的大小，集群所需的时间会有所不同。

2.2 使用仅索引扫描

到目前为止，你已经看到何时使用索引，何时不使用索引。除此以外，还讨论了位图扫描。

然而，索引还有更多的内容。下面的两个例子只是略有不同，尽管性能差异可能相当大。下面是第一个查询。

```
test=# EXPLAIN SELECT * FROM t_test WHERE id = 34234;
QUERY PLAN
-----
Index Scan using idx_id on t_test
(cost=0.43..8.45 rows=1 width=9)
Index Cond: (id = 34234)
```

这里没有什么异常。PostgreSQL使用一个索引来查找单行。如果只选择单列，会发生什么？

```
test=# EXPLAIN SELECT id FROM t_test WHERE id = 34234;
QUERY PLAN
-----
Index Only Scan using idx_id on t_test
(cost=0.43..8.45 rows=1 width=4)
Index Cond: (id = 34234)
(2 rows)
```

正如你所看到的，计划已经从索引扫描变成了只用索引扫描。在我们的例子中，`id`列已经被建立了索引，所以它的内容自然在索引中。如果所有的数据已经可以从索引中取出，那么在大多数情况下就没有必要再去表上。只有在查询额外的字段时才需要到表中去，而这里不是这样的情况。因此，只用索引的扫描将保证比正常的索引扫描有更好的性能。

实际上，在这里和那里，在索引中包括一个额外的列，以享受这个功能的好处，甚至是很有意义的。在MSSQL中，添加额外的列被称为覆盖索引。从PostgreSQL 11开始，我们也有同样的功能，它使用`CREATE INDEX`中的`INCLUDE`关键字。

3 了解额外的B-树功能

在PostgreSQL中，索引是一个很大的领域，涵盖了数据库工作的许多方面。正如我在本书中已经概述过的，索引是性能的关键。没有适当的索引就没有好的性能。因此，值得检查一下，我们将在下面的小节中详细介绍的与索引有关的功能。

3.1 组合索引

在我作为一个专业的PostgreSQL支持供应商的工作中，我经常被问到关于组合索引和单独索引的区别。在本节中，我将尝试对这个问题进行一些说明。一般的规则是，如果单个索引可以回答你的问题，它通常是最好的选择。然而，你不能为人们过滤的所有可能的字段组合建立索引。你可以做的是利用组合索引的属性来实现尽可能多的收益。

假设我们有一个包含三列的表：postal_code, last_name, and first_name。一个电话簿会像这样利用一个组合索引。你会看到，数据是按地点排序的。在同一地点内，数据将按姓和名进行排序。

下表显示了在三列索引的情况下，哪些操作是可能的。

Query	Possible	Remarks
postal_code = 2700 AND last_name = 'Schönig' AND first_name = 'Hans'	是的	这是该索引的理想用例。
postal_code = 2700 AND last_name = 'Schönig'	是的	无限制。
last_name = 'Schönig' ' AND postal_code = 2700	是的	PostgreSQL 将简单地交换条件
postal_code = 2700	是的	这就像在postal_code上建立索引一样；组合索引只是需要在磁盘上有更多的空间。
first_name = 'Hans'	是的，但用例不同	PostgreSQL不能再使用索引的排序属性。然而，在一些罕见的情况下（通常是有无数列的非常广泛的表），如果扫描整个索引和读取非常广泛的表一样便宜，PostgreSQL会扫描整个索引。

如果列被分开索引，你很可能最终会看到位图扫描。当然，一个单一的手工定制的索引会更好。

3.2 添加功能索引

到目前为止，你已经看到了如何对一个列的内容进行索引，因为它是。然而，这可能并不总是你真正想要的。因此，PostgreSQL允许`'./'`作为功能索引的创建。其基本思想非常简单：不是为一个值建立索引，而是将一个函数的输出存储在索引中。

下面的例子显示了如何对id列的余弦进行索引。

```
test=# CREATE INDEX idx_cos ON t_random (cos(id));
CREATE INDEX
test=# ANALYZE;
ANALYZE
```

你所要做的就是把函数放在列的清单上，然后你就完成了。当然，这不会对所有种类的函数都有效。只有当函数的输出是不可改变的时候才能使用，如下面的例子中所示。

```
test=# SELECT age('2010-01-01 10:00:00'::timestamptz);
          age
-----
 10 years 7 mons 11 days 14:00:00
(1 row)
```

像年龄这样的函数其实并不适合用于索引，因为它们的输出并不是恒定的。时间在继续，因此，age的输出也会改变。PostgreSQL将明确禁止那些有可能在相同输入的情况下改变其结果的函数。cos函数在这方面是没有问题的，因为一个数值的余弦在1000年后仍然是一样的。

为了测试这个索引，我写了一个简单的查询来显示将会发生什么。

```
test=# EXPLAIN SELECT * FROM t_random WHERE cos(id) = 10;
QUERY PLAN
-----
Index Scan using idx_cos on t_random (cost=0.43..8.45 rows=1 width=9)
  Index Cond: (cos((id)::double precision) = '10'::double precision)
(2 rows)
```

正如预期的那样，功能索引将像任何其他索引一样使用。

3.3 减少空间消耗

索引是很好的，其主要目的是尽可能地加快事情的进展。与所有的好东西一样，索引也是有代价的：空间消耗。为了发挥它的魔力，索引必须以一种有组织的方式存储数值。如果你的表包含1000万个整数值，属于该表的索引在逻辑上将包含这1000万个整数值，外加额外的开销。

一个B树将包含一个指向表中每一行的指针，因此它肯定不是免费的。要想知道一个索引需要多少空间，你可以使用\di+命令询问psql。

```
test=# \di+
List of relations
 Schema | Name | Type | owner | Table | Persistence | Size | Description
-----+-----+-----+-----+-----+-----+-----+
-- 
-
 public | idx_cos | index | hs | t_random | permanent | 86 MB |
 public | idx_id | index | hs | t_test | permanent | 86 MB |
 public | idx_name | index | hs | t_test | permanent | 26 MB |
 public | idx_random | index | hs | t_random | permanent | 86 MB |
(4 rows)
```

在我的数据库中，为了存储这些索引，已经烧掉了284MB的惊人的数量。现在，将其与底层表所消耗的存储量进行比较。

```

test=# \d+
List of relations
 Schema | Name | Type | Owner | Persistence | Size | Description
-----+-----+-----+-----+-----+-----+-----+
-- 
-
 public | t_random | table | hs | permanent | 169 MB |
 public | t_test | table | hs | permanent | 169 MB |
 public | t_test_id_seq | sequence | hs | permanent | 8192 bytes |
(3 rows)

```

这两个表的大小加起来也就338MB。换句话说，我们的索引需要的空间比实际数据要多。在现实世界中，这种情况很常见，实际上也很有可能。最近，我访问了德国的一个Cybertec客户，我看到了一个数据库，其中64%的数据库大小是由从未使用过的索引组成的（在几个月的时间里没有一次使用过）。所以，过度索引可能是一个问题，就像索引不足一样。记住，这些索引不只是消耗空间。每个INSERT或UPDATE函数也必须维护索引中的值。在极端情况下，比如我们的例子，这大大降低了写入的吞吐量。

如果表中只有少数不同的值，部分索引是一个解决方案。

```

test=# DROP INDEX idx_name;
DROP INDEX
test=# CREATE INDEX idx_name ON t_test (name)
 WHERE name NOT IN ('hans', 'paul');
CREATE INDEX

```

在下面这种情况下，大多数人已经被排除在索引之外，可以享受一个小而有效的索引。

```

test=# \di+ idx_name
List of relations
 Schema | Name | Type | Owner | Table | Persistence | Size | Description
-----+-----+-----+-----+-----+-----+-----+
-- 
-
 public | idx_name | index | hs | t_test | permanent | 8192 bytes |
(1 row)

```

请注意，只有排除构成表的大部分（至少25%左右）的非常频繁的值才有意义。部分索引的理想候选者是性别（我们假设大多数人是男性或女性）、国籍（假设你所在国家的大多数人有相同的国籍），等等。当然，应用这种技巧需要对你的数据有一些深入的了解，但它肯定会有回报。

3.4 索引时添加数据

创建一个索引很容易。然而，请记住，在建立索引的过程中，你不能修改表。CREATE INDEX命令将使用SHARE锁来锁定表，以确保没有变化发生。虽然这对小表来说显然没有问题，但对生产系统中的大表来说会造成问题。为1TB左右的数据建立索引需要一定的时间，因此封锁表的时间过长会成为一个问题。

解决这个问题的方法是CREATE INDEX CONCURRENTLY命令。构建索引将花费更多的时间（通常，至少是两倍），但是在创建索引期间，你可以正常使用表。下面是它的工作原理。

```
test=# CREATE INDEX CONCURRENTLY idx_name2 ON t_test (name);
CREATE INDEX
```

注意，如果你使用CREATE INDEX CONCURRENTLY命令，PostgreSQL并不保证成功。如果在你的系统上运行的操作与索引的创建有某种程度的冲突，那么索引最终可能被标记为无效的。如果你想弄清楚你的索引是否无效，在关系上使用\di。

4 介绍运算符类

到目前为止，我们的目标是弄清楚要索引什么，以及是否要盲目地在这一列或一组列上应用索引。然而，有一个假设，我们已经默默地接受了，以使其发挥作用。到目前为止，我们的工作假设是，数据必须被排序的顺序是一个有点固定的常数。在现实中，这个假设可能不成立。当然，数字总是以相同的顺序排列，但其他类型的数据很可能没有预定义的、固定的排序顺序。

为了证明我的观点，我编了一个真实世界的例子。看一下下面两条记录。

```
1118 09 08 78
2345 01 05 77
```

我现在的问题是，这两行的顺序是否正确？它们可能是，因为一个在另一个之前。然而，这是不对的，因为这两行确实有一些隐藏的语义。你在这里看到的是两个奥地利的社会安全号码。09 08 78实际上意味着1978年8月9日，而01 05 77实际上意味着1977年5月1日。前四个数字由一个校验和和某种自动递增的三位数组成。因此，在现实中，1977年在1978年之前，我们可以考虑将这两行互换，以达到理想的排序顺序。

问题是，PostgreSQL不知道这两行到底是什么意思。如果一个列被标记为文本，PostgreSQL将应用标准规则对文本进行排序。如果列被标记为数字，PostgreSQL将应用标准规则对数字进行排序。在任何情况下，它都不会使用我所描述的那样奇怪的东西。如果你认为我之前概述的事实是处理这些数字时唯一需要考虑的事情，那你就错了。一年有多少个月？12？远非如此，在这种情况下。在奥地利的社会保障体系中，这些数字最多可容纳14个月。为什么？请记住……三位数只是一个自动递增值。麻烦的是，如果一个移民或难民没有有效的文件，如果他们的生日不知道，他们将被分配一个人为的生日在第13个月。在1990年的巴尔干战争期间，奥地利向超过11.5万名难民提供了庇护。自然，这个三位数的数字是不够的，于是又增加了第14个月。现在，哪种标准数据类型可以处理这种1970年代初遗留下来的COBOL（也就是社会保障号码的布局被引入的时候）？答案是没有。

为了以理智的方式处理特殊用途的字段，PostgreSQL提供了运算符类。

```
test=# \h CREATE OPERATOR CLASS
Command: CREATE OPERATOR CLASS
Description: define a new operator class
Syntax:
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
  USING index_method [ FAMILY family_name ] AS
  { OPERATOR strategy_number operator_name [ ( op_type, op_type ) ]
    [ FOR SEARCH | FOR ORDER BY sort_family_name ]
    | FUNCTION support_number [ ( op_type [ , op_type ] ) ] function_name (
      argument_type [ , ... ] )
    | STORAGE storage_type
  } [ , ... ]
```

一个操作符类将告诉一个索引如何行动。让我们看一下一个标准的二叉树。它可以执行五个操作。

Strategy	Operator	Description
1	<	小于
2	<=	小于等于
3	=	等于
4	>=	大于等于
5	>	大于

标准运算符类支持我们在本书中使用的标准数据类型和标准运算符。如果你想处理社会安全号码，就有必要想出你自己的运算符，能够为你提供你所需要的逻辑。然后，这些自定义的运算符可以用来组成一个运算符类，这只不过是一个策略，它被传递给索引，以配置它应该如何行动。

4.1 为 B 树创建运算符类

为了给你一个操作者类的实际例子，我编写了一些代码来处理社会安全号码。为了保持简单，我没有注意到诸如校验之类的细节。

4.2 创建新的运算符

我们必须做的第一件事是想出所需的运算符。请注意，需要五个运算符。每个策略都有一个运算符。一个索引的策略其实就像一个插件，允许你放入自己的代码。在开始之前，我已经编译了一些测试数据。

```
CREATE TABLE t_sva (sva text);
INSERT INTO t_sva VALUES ('1118090878');
INSERT INTO t_sva VALUES ('2345010477');
```

现在测试数据已经有了，是时候创建一个操作符了。为了这个目的，PostgreSQL提供了CREATE OPERATOR命令。

```
test=# \h CREATE OPERATOR
Command: CREATE OPERATOR
Description: define a new operator
Syntax:
CREATE OPERATOR name (
    PROCEDURE = function_name
    [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
)
URL: https://www.postgresql.org/docs/13/sql-createoperator.html
```

基本上，其概念如下：运算符调用一个函数，该函数得到一个或两个参数；一个是运算符的左参数，一个是右参数。

正如你所看到的，操作符只不过是一个函数调用。因此，有必要在那些被运算符隐藏的函数中实现所需的逻辑。为了固定排序顺序，我写了一个名为normalize_si的函数。

```
CREATE OR REPLACE FUNCTION normalize_si(text) RETURNS text AS $$  
BEGIN  
    RETURN substring($1, 9, 2) ||  
           substring($1, 7, 2) ||  
           substring($1, 5, 2) ||  
           substring($1, 1, 4);  
END; $$  
LANGUAGE 'plpgsql' IMMUTABLE;
```

调用此函数将返回以下结果：

```
test=# SELECT normalize_si('1118090878');  
normalize_si  
-----  
7808091118  
(1 row)
```

正如你所看到的，我们所做的只是交换了一些数字。现在可以直接使用正常的字符串排序顺序。在下一步，这个函数已经可以用来直接比较社会安全号码。

第一个需要的函数是小于函数，这是第一个策略所需要的。

```
CREATE OR REPLACE FUNCTION si_lt(text, text) RETURNS boolean AS $$  
BEGIN  
    RETURN normalize_si($1) < normalize_si($2);  
END;  
$$ LANGUAGE 'plpgsql' IMMUTABLE;
```

这里有两件重要的事情要注意。

- 该函数不能用SQL语言编写。它只能在程序性语言或编译语言中工作。这样做的原因是，SQL函数在某些情况下可以被内联，而这将削弱整个工作。
- 第二个问题是，你应该坚持本章中使用的命名规则--它被社区广泛接受。Lessthan函数应该被称为`lt`，小于或等于的函数应该被称为`le`，以此类推。

鉴于这些知识，我们未来的操作者需要的下一组函数可以定义如下。

```
-- lower equals  
CREATE OR REPLACE FUNCTION si_le(text, text)  
RETURNS boolean AS  
$$  
BEGIN  
    RETURN normalize_si($1) <= normalize_si($2);  
END;  
$$  
LANGUAGE 'plpgsql' IMMUTABLE;  
-- greater equal
```

```

CREATE OR REPLACE FUNCTION si_ge(text, text)
RETURNS boolean AS
$$
BEGIN
RETURN normalize_si($1) >= normalize_si($2);
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;
-- greater
CREATE OR REPLACE FUNCTION si_gt(text, text)
RETURNS boolean AS
$$
BEGIN
RETURN normalize_si($1) > normalize_si($2);
END;
$$
LANGUAGE 'plpgsql' IMMUTABLE;

```

到目前为止，已经定义了四个函数。没有必要为等价运算符设置第五个函数。我们可以简单地采用现有的运算符，因为无论如何，equals并不依赖于排序顺序。现在所有的函数都到位了，是时候定义这些运算符了。

```

-- define operators
CREATE OPERATOR <# ( PROCEDURE=si_lt,
LEFTARG=text,
RIGHTARG=text);

```

操作符的设计实际上非常简单。操作符需要一个名字（在我的例子中是<#），一个应该被调用的过程，以及左、右参数的数据类型。当操作符被调用时，左参数将是si_lt的第一个参数，而右参数将是第二个参数。其余三个运算符也遵循同样的原则。

```

CREATE OPERATOR <#= ( PROCEDURE=si_le,
LEFTARG=text,
RIGHTARG=text);
CREATE OPERATOR >#= ( PROCEDURE=si_ge,
LEFTARG=text,
RIGHTARG=text);
CREATE OPERATOR ># ( PROCEDURE=si_gt,
LEFTARG=text,
RIGHTARG=text);

```

根据你所使用的索引类型，需要几个支持函数。在标准B树的情况下，只需要一个支持函数，它是用来加快内部速度的。

```

CREATE OR REPLACE FUNCTION si_same(text, text) RETURNS int AS $$ 
BEGIN
IF normalize_si($1) < normalize_si($2)
THEN
RETURN -1;
ELSIF normalize_si($1) > normalize_si($2)
THEN
RETURN +1;
ELSE

```

```
RETURN 0;
END IF;
END;
$$ LANGUAGE 'plpgsql' IMMUTABLE;
```

si_same函数将在第一个参数较小的情况下返回-1，如果两个参数相等则返回0，如果第一个参数较大则返回1。在内部，_same函数是工作主力，所以你应该确保你的代码被优化。

4.3 创建运算符类

最后，所有的组件都到位了，就可以创建索引所需要的运算符类了。

```
CREATE OPERATOR CLASS sva_special_ops
FOR TYPE text USING btree
AS
OPERATOR 1 <# ,
OPERATOR 2 <=# ,
OPERATOR 3 = ,
OPERATOR 4 >=# ,
OPERATOR 5 ># ,
FUNCTION 1 si_same(text, text);
```

CREATE OPERATOR CLASS命令将策略和运算符与OPERATOR 1连接起来。<#表示策略1将使用<#运算符。最后，_相同的函数与运算符类相连接。

请注意，运算符类有一个名字，而且它已经被明确定义为与B树一起工作。操作符类已经可以在创建索引时使用了。

```
CREATE INDEX idx_special ON t_sva (sva sva_special_ops);
```

创建索引的工作方式与以前略有不同：sva sva_special_ops意味着sva列使用sva_special_ops运算符类进行索引。如果没有明确使用sva_special_ops，那么PostgreSQL将不会采用我们的特殊排序顺序，而会决定使用默认的运算符类。

4.4 测试自定义运算符类

在我们的例子中，测试数据只包括两行。因此，PostgreSQL不会使用索引，因为这个表太小了，甚至不值得打开索引的开销。为了能够在不加载太多数据的情况下仍然进行测试，你可以建议优化器让顺序扫描变得更昂贵。

使操作更昂贵可以在你的会话中使用下面的指令来完成。

```
SET enable_seqscan TO off;
```

该索引按预期工作：

```
test=# explain SELECT * FROM t_sva WHERE sva = '0000112273';
QUERY PLAN
```

```
Index Only Scan using idx_special on t_sva
(cost=0.13..8.14 rows=1 width=32)
Index Cond: (sva = '0000112273'::text)
(2 rows)
test=# SELECT * FROM t_sva;
sva
-----
2345010477
1118090878
(2 rows)
```

5 了解PostgreSQL索引类型

到目前为止，我们只讨论了二叉树。然而，在许多情况下，二叉树是不够的。这是为什么呢？正如我们在本章已经讨论过的，B树基本上是基于排序的。`<`, `<=`, `=`, `>=`, 和 `>` 操作符都可以用B树来处理。问题是，并不是每一种数据类型都能以一种有用的方式进行排序。试想一下，一个多边形。你如何以一种有用的方式对这些对象进行排序？当然，你可以按照覆盖的面积、长度等进行排序，但这样做并不能让你真正使用几何搜索找到它们。

解决这个问题的办法是提供不止一种索引类型。每个索引都会有一个特殊的用途，并准确地完成所需的任务。以下六种索引类型是可用的（截止到PostgreSQL 10.0）。

```
test=# SELECT * FROM pg_am;
oid | amname | amhandler | amtype
-----+-----+-----+-----+
2 | heap | heap_tableam_handler | t
403 | btree | bthandler | i
405 | hash | hashhandler | i
783 | gist | gishtandler | i
2742 | gin | ginhandler | i
4000 | spgist | spghandler | i
3580 | brin | brinhandler | i
(7 rows)
```

B-树已经被详细地讨论过了，但是那些其他的索引类型有什么用呢？下面的章节将概述PostgreSQL中每种索引类型的用途。请注意，在这里你可以看到的基础上，还有一些扩展可以使用。网络上可用的其他索引类型有高速全文检索、vodka，以及将来的cognac。

5.1 哈希索引

哈希索引已经存在了很多年了。其想法是对输入值进行散列，并将其存储起来，以便日后查询。拥有哈希索引实际上是有意义的。然而，在PostgreSQL 10.0之前，使用哈希索引是不可取的，因为PostgreSQL对它们没有WAL支持。在PostgreSQL 10.0中，这种情况已经改变。哈希索引现在是完全记录的，因此可以用于复制，并且被认为是100%崩溃安全的。

哈希索引通常比B树索引要大一些。假设你想为400万个整数值做索引。一个B树将需要大约90MB的存储空间来完成这个任务。哈希索引则需要125MB左右的磁盘空间。许多人的假设是，哈希在磁盘上是超级别的，在许多情况下，这个假设是错误的。

5.2 GiST 索引

广义搜索树 (GiST) 索引是一种非常重要的索引类型，因为它们被用于各种不同的事情。GiST索引可以用来实现R树的行为，甚至有可能作为B树的行为。然而，不建议将GiST滥用于B树的索引。

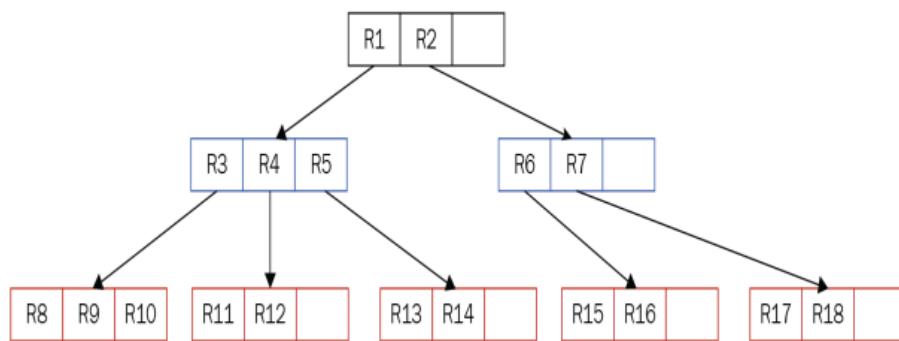
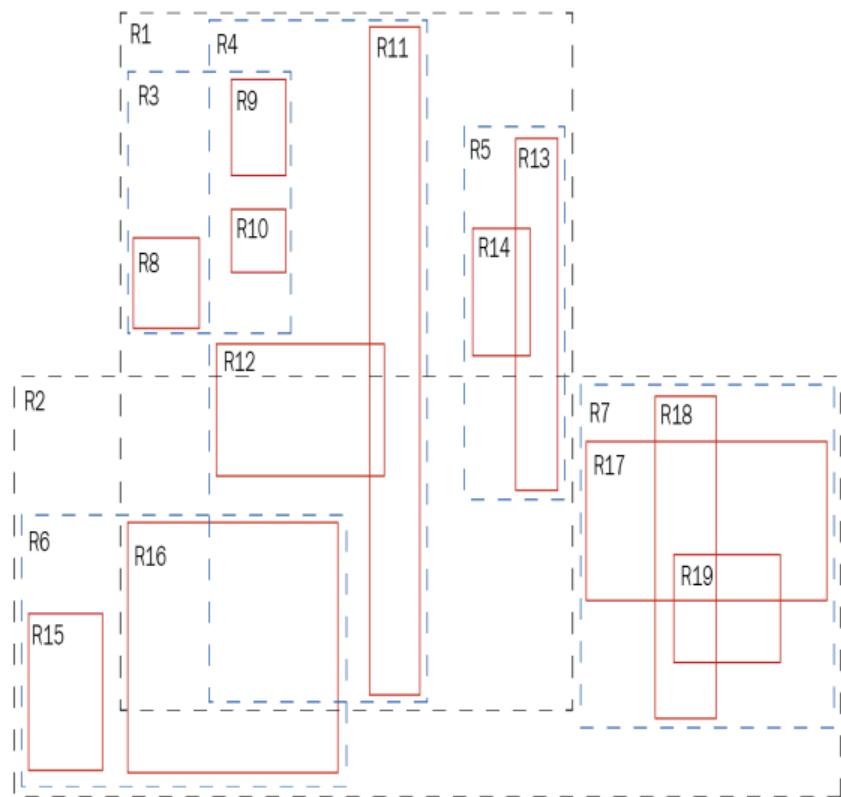
GiSTS的典型使用情况如下。

- 范围类型
- 几何索引 (例如，非常流行的PostGIS扩展所使用的那些)
- 模糊搜索

5.2.1 了解 GiST 的工作原理

对很多人来说，GiST仍然是一个黑盒子。因此，我决定在本章中增加一个章节，概述GiST的内部工作原理。

请看下面的图。



看一下这棵树。你可以看到，R1和R2在上面。R1和R2是包含其他东西的边界框。R3、R4和R5被R1所包含。R8、R9和R10被R3所包含，以此类推。因此，一个GiST索引是分层次组织的。在前面的图中你可以看到，一些在B树中没有的操作被支持。其中一些操作是重叠、左边、右边等等。GiST树的布局是几何索引的理想选择

5.2.2 扩展 GiST

当然，也可以想出自己的操作者类别。支持以下策略。

Operation	Operation Strategy number
Strictly left of	1
Does not extend to right of	2
Overlaps	3
Does not extend to left of	4
Strictly right of	5
Same	6
Contains	7
Contained by	8
Does not extend above	9
Strictly below	10
Strictly above	11
Does not extend below	12

如果你想为GiST写操作符类，必须提供几个支持函数。在B树的情况下，只有一个功能--GiST索引提供了更多的功能。

Function	Description	Support function number
consistent	这决定了一个键是否满足查询限定词。在内部，策略被查询和检查。	1
union	计算一组键的联合。在数值的情况下，计算上限和下限或一个范围。这对几何图形特别重要。	2
compress	这将计算出一个键或值的压缩表示。	3
decompress	这是与压缩功能相对应的函数	4
penalty	在插入过程中，将计算插入到树中的成本。该成本决定了新条目在树中的位置。因此，一个好的惩罚函数是从索引中获得良好整体性能的关键。	5
picksplit	决定在页面分割的情况下，将条目移到哪里。一些条目必须留在旧的页面上，而其他条目将被移到正在创建的新页面上。拥有一个好的picksplit功能对于提供良好的索引性能是至关重要的。	6
equal	等价函数与你已经在B树中看到的函数类似	7
distance	计算一个键和查询值之间的距离（一个数字）。距离函数是8个可选的，如果支持k-NN搜索，就需要这个函数。	8
fetch	确定一个压缩键的原始表示。这个函数需要用来处理仅有索引的扫描，正如PostgreSQL的最新版本所支持的那样。	9

如果你对一个好的例子感兴趣，我建议你去看看contrib目录下的btree_GiST模块。它展示了如何使用GiST对标准数据类型进行索引，是一个很好的信息来源，同时也是一个灵感来源。

5.3 GIN 索引

广义反转 (GIN) 索引是一种很好的文本索引方式。假设你想为100万个文本文件编制索引。某一个词可能会出现数百万次。在一个普通的B树中，这将意味着键被存储了数百万次。在GIN中，情况并非如此。每个键（或词）被存储一次，并分配给一个文档列表。键被组织在一个标准的Btree中。每个条目都会有一个文档列表，指向表中具有相同键的所有条目。GIN索引非常小而紧凑。然而，它缺少一个在B树中发现的重要特征--排序数据。在GIN中，与某一键相关的项目指针列表是按照该行在表中的位置排序的，而不是按照一些任意的标准。

5.3.1 扩展 GIN

就像任何其他指数一样，GIN可以被扩展。有以下策略可供选择。

Operation	Strategy number
Overlap	1
Contains	2
Is contained by	3
Equal	4

在此基础上，还提供了以下支持功能。

Function	Description	Support function number
compare	这个函数与你在B树中看到的函数相似。如果两个键被比较，它返回-1（低），0（相等）或1（高）。	1
extractValue	从一个要被索引的值中提取键。一个值可以有许多键。例如，一个文本值可能由一个以上的词组成	2
extractQuery	从查询条件中提取键	3
consistent	检查值是否匹配查询条件。	4
comparePartial	比较来自查询的部分键和来自索引的一个键。返回-1、0或1（类似于B-树支持的相同功能）。	5
triConsistent	判断一个值是否符合查询条件（三元变体）。如果存在一致函数，它是可选的。	6

如果你正在寻找一个如何扩展GIN的好例子，可以考虑看一下PostgreSQL contrib目录下的btree_gin模块。它是一个有价值的信息来源，也是你开始自己的实现的一个好方法。

如果你对全文搜索感兴趣，更多信息将在本章的理解全文搜索部分提供

5.4 SP-GiST 索引

空间分区的GiST (SP-GiST) 主要是为内存中的使用而设计的。其原因是，存储在磁盘上的SP-GiST需要相当多的磁盘点击才能发挥作用。磁盘点击比在RAM中跟踪几个指针要昂贵得多。

美中不足的是，SP-GiST可以用来实现各种类型的树，如四叉树、k-d树和弧形树（tries）。

以下是提供的策略

Operation	Strategy number
Strictly left of	1
Strictly right of	5
Same	6
Contained by	8
Strictly below	10
Strictly above	11

要为SP-GiST编写你自己的操作者类，必须提供几个函数。

Function	Description	Support function number
config	提供有关正在使用的运算符类的信息	1
choose	弄清楚如何将新值插入内部元组	2
picksplit	弄清楚如何划分/拆分一组值	3
inner_consistent	确定哪些子区需要被搜索到，以进行查询	4
leaf_consistent	确定键是否满足查询限定符	5

5.5 BRIN索引

区块范围索引 (BRINs) 有很大的实际用途。到目前为止，我们所讨论的所有索引都需要相当多的磁盘空间。尽管在缩小GIN索引等方面做了很多工作，但它们仍然需要相当多的磁盘空间，因为每个条目都需要一个索引指针。因此，如果有1000万个条目，就会有1000万个索引指针。空间是BRINs所要解决的主要问题。一个BRIN不为每个元组保留一个索引条目，但会存储128个（默认）数据块（1MB）的最小值和最大值。因此，该索引非常小，但有损失。扫描该索引将返回比我们要求的更多的数据。

PostgreSQL必须在后面的步骤中过滤掉这些额外的行。

下面的例子说明了BRIN到底有多小。

```
test=# CREATE INDEX idx_brin ON t_test USING brin(id);
CREATE INDEX
test=# \di+ idx_brin
List of relations
 Schema | Name | Type | Owner | Table | Persistence | Size | Description
-----+-----+-----+-----+-----+-----+-----+
 public | idx_brin | index | hs | t_test | permanent | 48 kB |
(1 row)
```

在我的例子中，BRIN索引比标准Btree要小2000倍。现在自然产生的问题是，为什么我们不总是使用BRIN索引？要回答这类问题，重要的是要反思BRIN的布局；1MB的最小值和最大值被存储。如果数据是排序的（高相关性），BRIN是相当高效的，因为我们可以获取1MB的数据并进行扫描，就可以了。然而，如果数据被洗牌了呢？在这种情况下，BRIN就不能再排除大块的数据了，因为很可能在这1MB的数据中就有接近整体高点和整体低点的东西。因此，BRIN主要是为高度相关的数据制作的。在现实中，相关的数据在数据仓库应用中是很有可能的。通常情况下，数据每天都在加载，因此日期可能是高度相关的。

5.5.1 扩展 BRIN 索引

BRIN支持与B-tree相同的策略，因此需要相同的运算符集。这些代码可以被很好地重复使用。

Operation	Strategy number
Less than	1
Less than or equal	2
Equal	3
Greater than or equal	4
Greater than	5

BRIN需要的支持功能如下：

Function	Description	Support function number
opclInfo	提供有关索引列的内部信息	1
add_value	在一个现有的摘要元组中添加一个条目	2
consistent	检查值是否符合条件	3
union	计算两个摘要条目的联合（最小/最大值）。	4

5.6 添加额外的索引

从PostgreSQL 9.6开始，就有了一种简单的方法来部署全新的索引类型作为扩展。这非常酷，因为如果PostgreSQL提供的那些索引类型不够，可以增加额外的索引类型来满足你的目的。这样做的指令是
`CREATE ACCESS METHOD`。

```
test=# \h CREATE ACCESS METHOD
Command: CREATE ACCESS METHOD
Description: define a new access method
Syntax:
CREATE ACCESS METHOD name
    TYPE access_method_type
    HANDLER handler_function
URL: https://www.postgresql.org/docs/13/sql-create-access-method.html
```

不要太担心这个命令--如果你有一天部署了你自己的索引类型，它将作为一个随时可用的扩展出现。

其中一个扩展实现了布隆过滤器。布隆过滤器是概率数据结构。它们有时会返回太多的行，但绝不会太少。因此，布隆过滤器是一种预过滤数据的好方法。

它是如何工作的？布隆过滤器是在几个列上定义的。根据输入值计算出一个位掩码，然后将其与你的查询进行比较。布隆过滤器的好处是，你可以对你想要的列进行索引。缺点是必须要读取整个布隆过滤器。当然，布隆过滤器比底层数据要小，所以在很多情况下，它是非常有益的。

要使用布隆过滤器，只需激活这个扩展，它是PostgreSQL contrib包的一部分。

```
test=# CREATE EXTENSION bloom;
CREATE EXTENSION
```

正如我们之前所说的，布隆过滤器背后的想法是，它允许你为你想要的许多列做索引。在许多现实世界的应用中，面临的挑战是在不知道用户在运行时实际需要哪些组合的情况下对许多列进行索引。在一个大表的情况下，完全不可能在比如说80个字段或更多的字段上创建标准的B树索引。在这种情况下，布隆过滤器可能是一个替代方案。

```
test=# CREATE TABLE t_bloom (x1 int, x2 int, x3 int, x4 int,
   x5 int, x6 int, x7 int);
CREATE TABLE
```

创建索引很简单：

```
test=# CREATE INDEX idx_bloom ON t_bloom USING bloom(x1, x2, x3, x4,  
x5, x6, x7);  
CREATE INDEX
```

如果关闭顺序扫描，就可以看到索引的作用。

```
test=# SET enable_seqscan TO off;  
SET  
test=# explain SELECT * FROM t_bloom WHERE x5 = 9 AND x3 = 7;  
QUERY PLAN  
-----  
Bitmap Heap Scan on t_bloom (cost=18.50..22.52 rows=1 width=28)  
Recheck Cond: ((x3 = 7) AND (x5 = 9))  
-> Bitmap Index Scan on idx_bloom (cost=0.00..18.50 rows=1 width=0)  
Index Cond: ((x3 = 7) AND (x5 = 9))
```

请注意，我查询的是一个随机列的组合；它们与索引中的实际顺序没有关系。布隆过滤器仍将是有益的。

6 用模糊搜索实现更好的答案

如今，执行精确的搜索并不是用户唯一期望的事情。现代网站对用户的教育方式是，无论用户输入什么，他们总是期待一个结果。如果你在谷歌上搜索，总会有一个答案，即使用户的输入是错误的，充满了错别字，或者根本毫无意义。人们期望有好的结果，不管输入的数据是什么。

6.1 利用 pg_trgm

要用PostgreSQL做模糊搜索，你可以添加pg_trgm扩展。要激活这个扩展，只需运行以下命令。

```
test=# CREATE EXTENSION pg_trgm;  
CREATE EXTENSION
```

pg_trgm扩展功能相当强大，为了向你展示它的能力，我汇编了一些样本数据，包括奥地利这里的2354个村庄和城市的名称。

我们的样本数据可以存储在一个简单的表中。

```
test=# CREATE TABLE t_location (name text);  
CREATE TABLE
```

我的公司网站有所有的数据，PostgreSQL允许你直接加载数据。

```
test=# COPY t_location FROM PROGRAM  
'curl https://www.cybertec-postgresql.com/secret/orte.txt';  
COPY 2354
```

curl (一种获取数据的命令行工具) 必须安装。如果你没有这个工具, 可以正常下载文件并从你的本地文件系统中导入。

一旦数据被加载, 就可以检查出表的内容。

```
test=# SELECT * FROM t_location LIMIT 4;
name
-----
Eisenstadt
Rust
Breitenbrunn am Neusiedler See
Donnerskirchen
(4 rows)
```

如果德语不是你的母语, 可能无法在不出现严重错误的情况下拼出这些地点的名称。pg_trgm为我们提供了一个距离运算符, 计算两个字符串之间的距离。

```
test=# SELECT 'abcde' <-> 'abdeacb';
?column?
-----
0.833333
(1 row)
```

该距离是一个介于0和1之间的数字, 数字越小, 两个字符串就越相似。这是如何做到的? 三角形将一个字符串剖析成每三个字符的序列。

```
test=# SELECT show_trgm('abcdef');
show_trgm
-----
{" a"," ab",abc,bcd,cde,def,"ef "}
(1 row)
```

然后这些序列将被用来得出你刚才看到的距离。当然, 距离运算符可以在一个查询中使用, 以找到最接近的匹配。

```
test=# SELECT *
  FROM t_location
 ORDER BY name <-> 'Kramertneusiedel'
LIMIT 3;
name
-----
Gramatneusiedl
Klein-Neusiedl
Potzneusiedl
(3 rows)
```

Gramatneusiedl与Kramertneusiedel相当接近。它听起来很相似, 用K而不是G是一个很常见的错误。在谷歌上, 你有时会看到Did you mean...? 这很有可能是谷歌在这里使用了ngrams来做这个。在PostgreSQL中, 可以使用GiST使用三元组对文本进行索引。

```
test=# CREATE INDEX idx_trgm ON t_location
    USING GIST(name GIST_trgm_ops);
CREATE INDEX
```

pg_trgm为我们提供了GiST_trgm_ops操作者类，它是用来做相似度搜索的。下面的代码显示，索引的使用符合预期。

```
test=# explain SELECT *
  FROM t_location
 ORDER BY name <-> 'Kramertneusiedel'
 LIMIT 5;
QUERY PLAN

Limit (cost=0.14..0.58 rows=5 width=17)
-> Index Scan using idx_trgm on t_location
(cost=0.14..207.22 rows=2354 width=17)
Order By: (name <-> 'Kramertneusiedel'::text)
(3 rows)
```

在这个例子中，我使用了一个GiST索引来索引三元祖。然而，对于大型数据集来说，使用GIN代替要好得多。我写了一篇博文，说明其中的区别。你可以在我的网站上找到它，网址是<https://www.cybertec-postgresql.com/en/postgresql-more-performance-for-like-and-ilike-statements/>。

6.2 加快 LIKE 查询

LIKE 查询肯定会导致当今全球人们面临的一些最糟糕的性能问题。在大多数数据库系统中，LIKE的速度相当慢，而且需要进行顺序扫描。除此之外，终端用户很快就会发现，在很多情况下，模糊搜索会比精确查询返回更好的结果。因此，在一个大表上的单一类型的LIKE查询，如果被频繁调用，往往会削弱整个数据库服务器的性能。

幸运的是，PostgreSQL为这个问题提供了一个解决方案，而且这个解决方案刚好已经安装好了。

```
test=# explain SELECT * FROM t_location WHERE name LIKE '%neusi%';
QUERY PLAN

Bitmap Heap Scan on t_location
(cost=4.33..19.05 rows=24 width=13)
Recheck Cond: (name ~ '%neusi%'::text)
-> Bitmap Index Scan on idx_trgm (cost=0.00..4.32 rows=24 width=0)
Index Cond: (name ~ '%neusi%'::text)
(4 rows)
```

我们在上一节部署的三元祖索引也适用于加快LIKE的速度。请注意，%符号可以在搜索字符串的任何位置使用。这比标准的B-树有很大的优势，B-树只是碰巧在查询的末尾加快通配符的速度

6.3 处理正则表达式

然而，这仍然不是全部。Trigram索引甚至能够加快简单的正则表达式的速度。下面的例子显示了如何做到这一点。

```
test=# SELECT * FROM t_location WHERE name ~ '[A-C].*neu.*';
          name
-----
 Bruckneudorf
(1 row)
test=# explain SELECT * FROM t_location WHERE name ~ '[A-C].*neu.*';
      QUERY PLAN
-----
 Index Scan using idx_trgm on t_location (cost=0.14..8.16
 rows=1 width=13)
 Index Cond: (name ~ '[A-C].*neu.*'::text)
(2 rows)
```

PostgreSQL将检查正则表达式并使用索引来回答问题。

在内部，PostgreSQL可以将正则表达式转换成图，并相应地遍历索引。

7 了解全文搜索

如果你在查询名字或寻找简单的字符串，你通常是在查询一个字段的全部内容。在全文搜索中，这是不一样的。全文搜索的目的是寻找文本中可以找到的词或词组。因此，全文搜索更多的是一种包含操作，因为你基本上没有在寻找一个精确的字符串。

在PostgreSQL中，全文搜索可以使用GIN索引来完成。这个想法是剖析一个文本，提取有价值的词组（="预处理过的词组"）字符串，并对这些元素而不是基础文本进行索引。为了使你的搜索更加成功，这些词是经过预处理的。

下面是一个例子。

```
test=# SELECT to_tsvector('english',
  'A car, I want a car. I would not even mind
 having many cars');
          to_tsvector
-----
 'car':2,6,14 'even':10 'mani':13 'mind':11 'want':4 'would':8
(1 row)
```

这个例子显示了一个简单的句子。to_tsvector函数将接受这个字符串，应用英语规则，并执行一个词干处理过程。基于配置（english），PostgreSQL将解析该字符串，扔掉停止词，并对单个单词进行词干处理。例如，car和cars将被转换为car。请注意，这并不是要找到词干。在很多的情况下，PostgreSQL将简单地通过应用标准的规则将字符串转换为mani，这些规则与英语语言很好地配合。

请注意，to_tsvector函数的输出是高度依赖语言的。如果你告诉PostgreSQL把这个字符串当作荷兰语，结果将完全不同。

```
test=# SELECT to_tsvector('dutch', 'A car, I want a car. I would not even mind
having many cars');
          to_tsvector
-----
 'a':1,5 'car':2,6,14 'even':10 'having':12 'i':3,7 'many':13
'mind':11 'not':9 'would':8
(1 row)
```

要弄清哪些配置是被支持的，可以考虑运行以下查询。

```
SELECT cfgname FROM pg_ts_config;
```

现在让我们看看如何比较字符串。

7.1 比较字符串

在简要了解了词干提取过程之后，是时候弄清楚如何将词干提取文本与用户查询进行比较了。以下代码片段检查 wanted 一词：

```
test=# SELECT to_tsvector('english', 'A car, I want a car. I would not even mind
having many cars') @@ to_tsquery('english', 'wanted');
?column?
-----
 t
(1 row)
```

注意，想要的东西实际上并没有在原文中显示出来。但是，PostgreSQL仍然会返回true。原因是want 和wanted都被转换为同一个词素，所以结果是true。实际上，这有很大的意义。想象一下，你正在谷歌上寻找一辆汽车。如果你找到了卖汽车的网页，这就完全可以了。因此，寻找共同的词素是一个明智的想法。

有时，人们不只是在寻找一个词，而是想找到一组词。使用to_tsquery，这是有可能的，如下面的例子所示。

```
test=# SELECT to_tsvector('english', 'A car, I want a car. I would not even mind
having many cars') @@ to_tsquery('english', 'wanted & bmw');
?column?
-----
 f
(1 row)
```

在这种情况下，会返回false，因为在我们的输入字符串中找不到bmw。在to_tsquery函数中，&表示和，|表示或。因此，建立复杂的搜索字符串是很容易的。

7.2 定义 GIN 索引

如果你想在一列或一组列上应用文本搜索，基本上有两个选择。

- 使用GIN创建一个功能索引。
- 添加一个包含随时可用的tsvector对象的列和一个触发器来保持它们的同步。

在这一节中，将对这两种选择进行概述。为了向你展示事情是如何进行的，我已经创建了一些样本数据。

```
test=# CREATE TABLE t_fts AS SELECT comment
  FROM pg_available_extensions;
SELECT 43
```

用功能索引直接为列建立索无疑是一种较慢但更节省空间的方式。

```
test=# CREATE INDEX idx_fts_func ON t_fts
  USING gin(to_tsvector('english', comment));
CREATE INDEX
```

在函数上部署索引很容易，但会导致一些开销。添加一个物化列需要更多空间，但会导致更好的运行时行为。

```
test=# ALTER TABLE t_fts ADD COLUMN ts tsvector;
ALTER TABLE
```

唯一的麻烦是，你如何让这一列保持同步？答案是通过使用一个触发器。

```
test=# CREATE TRIGGER tsvectorupdate
  BEFORE INSERT OR UPDATE ON t_fts
  FOR EACH ROW
  EXECUTE PROCEDURE
    tsvector_update_trigger(somename, 'pg_catalog.english', 'comment');
```

幸运的是，PostgreSQL已经提供了一个C函数，可以被触发器用来同步tsvector列。只要给函数传递一个名称、所需的语言和几个列，你就已经完成了。触发器函数将处理所有需要的事情。注意，一个触发器总是在与进行修改的语句相同的事务中操作。因此，不存在不一致的风险。

7.3 调试搜索

有时，不太清楚查询与给定搜索字符串匹配的原因。为了调试您的查询，PostgreSQL 提供了 `ts_debug` 函数。从用户的角度来看，它可以像 `to_tsvector` 一样使用。它揭示了很多关于全文搜索基础设施的内部工作原理：

```
test=# \x
Expanded display is on.
test=# SELECT * FROM ts_debug('english', 'go to www.cybertec-postgresql.com');
-[ RECORD 1 ]+-----
alias | asciiword
description | word, all ASCII
token | go
dictionaries | {english_stem}
dictionary | english_stem
lexemes | {go}
-[ RECORD 2 ]+-----
alias | blank
description | space symbols
```

```
token |
dictionaries | {}
dictionary |
lexemes |
-[ RECORD 3 ]+-----
alias | asciiword
description | word, all ASCII
token | to
dictionaries | {english_stem}
dictionary | english_stem
lexemes | {}
-[ RECORD 4 ]+-----
alias | blank
description | Space symbols
token |
dictionaries | {}
dictionary |
lexemes |
-[ RECORD 5 ]+-----
alias | host
description | Host
token | www.cybertec-postgresql.com
dictionaries | {simple}
dictionary | simple
lexemes | {www.cybertec-postgresql.com}
```

ts_debug 会列出每一个找到的标记，并显示关于标记的信息。你会看到解析器找到了哪个标记，使用的字典，以及对象的类型。在我的例子中，发现了空白、单词和主机。你也可能看到数字、电子邮件地址，以及更多。根据字符串的类型，PostgreSQL 将以不同的方式处理事情。例如，阻止主机名和电子邮件地址是完全没有意义的。

7.4 收集单词统计信息

全文搜索可以处理大量的数据。为了让最终用户对他们的文本有更多的了解，PostgreSQL 提供了 ts_stat 函数，它返回一个单词列表。

```
SELECT * FROM ts_stat('SELECT to_tsvector('''english''', comment)
FROM pg_available_extensions')
ORDER BY 2 DESC
LIMIT 3;
word | ndoc | nentry
-----+-----+
function | 10 | 10
data | 10 | 10
type | 7 | 7
(3 rows)
```

词列包含了干系词；ndoc 告诉我们某个词出现在多少个文档中。

7.5 利用排除运算符

到目前为止，索引一直被用来加快速度和确保唯一性。然而，几年前，有人想出了用索引来做更多事情的想法。正如你在本章中所看到的，GiST 支持诸如相交、重叠、包含等操作。那么，为什么不使用这些操作来管理数据的完整性呢？

下面是一个例子。

```
test=# CREATE EXTENSION btree_gist;
test=# CREATE TABLE t_reservation (
    room int,
    from_to tsrange,
    EXCLUDE USING GIST (room with =,
    from_to with &&)
);
CREATE TABLE
```

EXCLUDE USING GiST子句定义了额外的约束。如果你正在销售房间，你可能想允许不同的房间在同一时间被预订。但是，你不希望在同一时期将同一个房间出售两次。在我的例子中，EXCLUDE子句是这样说的：如果一个房间在同一时间被预订了两次，应该弹出一个错误（from_to中的数据不能重叠&&如果它与同一个房间有关）。

以下两行不会违反约束条件

```
test=# INSERT INTO t_reservation
VALUES (10, '['"2017-01-01", "2017-03-03"]');
INSERT 0 1
test=# INSERT INTO t_reservation
VALUES (13, '['"2017-01-01", "2017-03-03"]');
INSERT 0 1
```

然而，下一个INSERT将导致违规，因为数据重叠了。

```
test=# INSERT INTO t_reservation
VALUES (13, '['"2017-02-02", "2017-08-14"]');
psql: ERROR: conflicting key value violates exclusion constraint
"t_reservation_room_from_to_excl"
DETAIL: Key (room, from_to)=(13, ["2017-02-02 00:00:00", "2017-08-14 00:00:00"])
conflicts with existing key (room, from_to)=(13, ["2017-01-01 00:00:00", "2017-
03-03
00:00:00"]).
```

排除运算符的使用非常有用，可以为你提供高度先进的处理完整性的手段。

8 总结

这一章是关于索引的。我们了解了PostgreSQL何时会决定一个索引以及存在哪些类型的索引。在仅仅使用索引的基础上，还可以实现你自己的策略，用自定义运算符和索引策略来加速你的应用程序。

对于那些真正想把事情做到极致的人来说，PostgreSQL提供了自定义访问方法。

在第4章，处理高级SQL，我们将讨论高级SQL。许多人不知道SQL的真正能力，因此，我将向你展示一些高效、更高级的SQL技术。

9 问题

- 索引总是能提高性能吗？
- 索引会使用大量的空间吗？
- 怎样才能找到丢失的索引？
- 索引可以并行构建吗？

处理高级SQL

1 介绍分组集

- 1.1 加载一些样品数据
- 1.2 应用分组集
- 1.3 调查性能
- 1.4 将分组集与FILTER子句结合在一起

2 使用有序集

3 了解假设的聚合

4 利用窗口功能和分析

- 4.1 分区数据
- 4.2 窗口内的数据排序
- 4.3 使用滑动窗口
- 4.4 了解 ROWS 和 RANGE 之间的细微差别
- 4.5 使用 EXCLUDE TIES 和 EXCLUDE GROUP 删除重复项
- 4.6 抽象窗口子句
- 4.7 使用板载窗口功能
 - 4.7.1 rank 和 dense_rank 函数
 - 4.7.2 ntile() 函数
 - 4.7.3 lead() 和 lag() 函数
 - 4.7.4 first_value(), nth_value(), and last_value() 函数
 - 4.7.5 row_number() 函数

5 编写你自己的聚合

- 5.1 创建简单聚合
- 5.2 添加对并行查询的支持
- 5.3 提高效率
- 5.4 编写假设聚合

6 总结

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-9-4-aggregation-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
-----+-----+-----+-----+
(4 rows)
```

```
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} | {0,0,1,1,2,2,2}
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`lead(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。现在的关键是，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 $2.50 + 4 \times 2.2$ 。然后，加入下一条线路，将增加 3.2×2.2 ，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。

日志文件和系统统计信息

1 收集运行时的统计数据

- 1.1 使用 PostgreSQL 系统视图
 - 1.1.1 检查实时流量
 - 1.1.2 检查数据库
 - 1.1.3 检查表
 - 1.1.4 理解 pg_stat_user_tables
- 1.2 深入研究索引
- 1.3 追踪后台工作者
- 1.4 跟踪、存档和流式传输
- 1.5 检查 SSL 连接
- 1.6 实时检查事务情况
- 1.7 跟踪 VACUUM 和 CREATE INDEX 进度
- 1.8 使用 pg_stat_statements

2 创建日志文件

- 2.1 配置 postgresql.conf 文件
- 2.2 定义日志的目的地和轮换
- 2.3 配置系统日志
- 2.4 记录慢速查询
- 2.5 定义记录内容和方式

3 总结

4 问题

在第4章 "处理高级SQL" 中，你了解了高级SQL和从不同角度看待SQL的方法。然而，数据库工作并不是由黑客攻击花哨的SQL组成。有时，它是关于保持事情以专业的方式运行。要做到这一点，密切关注系统统计、日志文件等是非常重要的。监控是专业地运行数据库的关键。幸运的是，PostgreSQL有许多功能可以帮助你监控你的数据库，你将在本章中学习如何使用它们。

在这一章中，你将了解到以下主题。

- 收集运行时的统计数据
- 创建日志文件
- 收集重要的信息
- 了解数据库的统计数据

在本章结束时，你将能够正确地配置PostgreSQL的日志基础设施，并以最专业的方式处理日志文件。

1 收集运行时的统计数据

你真正需要学习的第一件事是了解PostgreSQL的内部统计有哪些功能，以及如何使用它们。在我看来，如果不收集必要的数据来做出谨慎的决定，就没有办法提高性能和可靠性。本节将引导你了解PostgreSQL的运行时统计，并解释你如何从数据库设置中提取更多的运行时信息。

1.1 使用 PostgreSQL 系统视图

PostgreSQL提供了大量的系统视图，允许管理员和开发人员深入了解他们系统中真正发生的事情。问题是，许多人实际上收集了所有这些数据，但却不能从中获得真正的意义。一般的规则是这样的：无论如何，为你不理解的东西绘制图表是没有意义的。因此，本节的目标是阐明PostgreSQL所提供的一些情况，希望能使用户更容易地利用那里的东西来使用。

1.1.1 检查实时流量

每当我检查一个系统来运行它，修复它，或者做一些其他的改进时，有一个系统视图我喜欢先检查，然后再深入挖掘。当然，我指的是pg_stat_activity。这个视图背后的想法是给你一个机会来弄清楚现在正在发生的事情。

下面是它的工作原理。

```
test=# \d pg_stat_activity
View "pg_catalog.pg_stat_activity"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----+
datid | oid | |||
datname | name | |||
pid | integer | |||
leader_pid | integer | |||
usesysid | oid | |||
username | name | |||
application_name | text | |||
client_addr | inet | |||
client_hostname | text | |||
client_port | integer | |||
backend_start | timestamp with time zone | |||
xact_start | timestamp with time zone | |||
query_start | timestamp with time zone | |||
state_change | timestamp with time zone | |||
wait_event_type | text | |||
wait_event | text | |||
state | text | |||
backend_xid | xid | |||
backend_xmin | xid | |||
query | text | |||
backend_type | text | |||
```

此外，pg_stat_activity将为你提供每个活动连接的一行信息。你将看到数据库的内部对象ID (datid)，某人所连接的数据库的名称，以及为这个连接服务的进程ID (pid)。除此之外，PostgreSQL会告诉你谁在连接 (username; 注意缺少r) 和该用户的内部对象ID (usesysid)。

然后，有一个叫做application_name的字段，这个字段值得更广泛地评论一下。一般来说，application_name可以由终端用户自由设置，如下所示。

```
test=# SET application_name TO 'www.cybertec-postgresql.com';
SET
test=# SHOW application_name;
 application_name
-----
 www.cybertec-postgresql.com
(1 row)
```

重点是：让我们假设有成千上万的连接来自一个IP。作为管理员，你能知道一个特定的连接现在到底在做什么吗？你可能不知道所有的SQL内容。如果客户端好心地设置了一个application_name参数，那就更容易看出一个连接的真正目的了。在我的例子中，我将名称设置为连接所属的域。这使得我们很容易找到可能导致类似问题的类似连接。

接下来的三列 (client_) 将告诉你一个连接来自哪里。PostgreSQL将显示IP地址和 (如果它被配置为)甚至主机名。

此外，backend_start将告诉你某个连接何时开始，xact_start表明一个事务何时开始。然后，还有query_start和state_change。在过去的黑暗时期，PostgreSQL只显示活动的查询。在那个查询时间比现在长很多的时代，这是有意义的。在现代硬件上，OLTP（在线事务处理）查询可能只消耗几分之一的时间，因此很难抓住这种查询的潜在危害。解决办法是显示正在进行的查询或你正在看的连接所执行的前一个查询。

以下是你可能看到的情况。

```
test=# SELECT pid, query_start, state_change, state, query
  FROM pg_stat_activity;
...
-[ RECORD 2 ] +-----
pid | 28001
query_start | 2020-09-05 10:03:57.575593+01
state_change | 2020-09-05 10:03:57.575595+01
state | active
query | SELECT pg_sleep(10000000);
```

在这种情况下，你可以看到pg_sleep正在第二个连接中执行。一旦这个查询被终止，输出就会改变，如下面的代码所示。

```
-[ RECORD 2 ]+-----
pid | 28001
query_start | 2020-09-05 10:03:57.575593+01
state_change | 2020-09-05 10:05:10.388522+01
state | idle
query | SELECT pg_sleep(10000000);
```

现在查询被标记为空闲。state_change和query_start的区别在于查询需要执行的时间。因此，pg_stat_activity会给你一个很好的概述，告诉你现在你的系统中正在发生什么。新的state_change字段使我们更有可能发现昂贵的查询。

现在的问题是：一旦你发现了不好的查询，你如何才能真正摆脱它们？PostgreSQL提供了两个函数来处理这些事情：

- pg_cancel_backend: pg_cancel_backend函数将终止查询，但会保留连接。
- pg_terminate_backend: pg_terminate_backend函数更激进一些，它将杀死整个数据库连接，以及查询。

如果你想断开除你自己之外的所有其他用户的连接，这里是你是如何做到的。

```
test=# SELECT pg_terminate_backend(pid)
  FROM pg_stat_activity
 WHERE pid <> pg_backend_pid()
   AND backend_type = 'client backend';
pg_terminate_backend

-----
t
t
(2 row)
```

我们为每一条符合WHERE条件的记录调用终止函数。

如果你碰巧被踢出，将显示以下信息。

```
test=# SELECT pg_sleep(10000000);
psql: FATAL: terminating connection due to administrator command server closed
the
connection unexpectedly
This probably means that the server terminated abnormally before or while
processing the request. The
connection to the server was lost. Attempting reset: succeeded.
```

只有psql会尝试重新连接。这对大多数其他客户端来说是不正确的 - 特别是对客户端库来说。

1.1.2 检查数据库

一旦你检查了活动的数据库连接，你就可以更深入地检查数据库级别的统计数据。pg_stat_database将在你的PostgreSQL实例中为每个数据库返回一行。

这就是你将在那里找到的东西。

```
test=# \d pg_stat_database
 View "pg_catalog.pg_stat_database"
 Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----+
-
-
datid | oid | |
datname | name | |
numbackends | integer | |
xact_commit | bigint | |
xact_rollback | bigint | |
blk_read | bigint | |
blk_hit | bigint | |
tup_returned | bigint | |
tup_fetched | bigint | |
tup_inserted | bigint | |
tup_updated | bigint | |
tup_deleted | bigint | |
conflicts | bigint | |
temp_files | bigint | |
temp_bytes | bigint | |
deadlocks | bigint | |
checksum_failures | bigint | |
checksum_last_failure | timestamp with time zone | |
blk_read_time | double precision | |
```

```
blk_write_time | double precision | | |
stats_reset | timestamp with time zone | | |
```

在数据库ID和数据库名称旁边有一列叫做numbackends，它显示了当前打开的数据库连接的数量。

然后，还有xact_commit和xact_rollback。这两列表明你的应用程序是否倾向于提交或回滚。blks_hit和blks_read将告诉你关于缓存的点击率和缓存的失误率。当检查这两列时，请记住，我们主要讨论的是共享缓冲区的点击率和共享缓冲区的失误。在数据库层面上，没有合理的方法来区分文件系统的缓存命中和真正的磁盘命中。在Cybertec (<https://www.cybertec-postgresql.com>)，我们喜欢在pg_stat_database中查看是否同时存在磁盘等待和缓存缺失，以了解系统中真正发生的情况。

tup_列会告诉你系统中是否有大量的读或大量的写正在进行。

然后，我们有temp_files和temp_bytes。这两列具有难以置信的重要性，因为它们将告诉你，你的数据库是否不得不向磁盘写入临时文件，这将不可避免地减慢操作。临时文件使用率高的原因是？主要原因如下。

- 设置不佳:如果你的work_mem设置太低，就没有办法在RAM中做任何事情，因此PostgreSQL会转到磁盘。
- 愚蠢的操作。经常发生的情况是，人们用相当昂贵和无意义的查询来折磨他们的系统。如果你在一个OLTP系统上看到许多临时文件，考虑检查一下昂贵的查询。
- 索引和其他管理任务。偶尔，索引可能会被创建，或者人们会运行DDLS。这些操作可能会导致临时文件I/O，但不一定被认为是一个问题（在许多情况下）。

简而言之，临时文件可能发生，即使你的系统完全正常。然而，密切关注它们并确保不经常需要临时文件是绝对有意义的。

最后，还有两个重要的字段：blk_read_time和blk_write_time。默认情况下，这两个字段是空的，没有数据被收集。这些字段背后的想法是让你看到有多少时间是花在I/O上的。这些字段为空的原因是track_io_timing默认为关闭。这是有原因的。想象一下，你想检查读取100万个块需要多长时间。要做到这一点，你必须调用C库中的时间函数两次，这导致了200万次额外的函数调用，只是为了读取8GB的数据。这真的取决于你的系统速度，因为这是否会导致大量的开销。

幸运的是，有一个工具可以帮助你确定计时的成本有多高，如下面的代码块所示。

```
[hs@zenbook ~]$ pg_test_timing
Testing timing overhead for 3 seconds.
Per loop time including overhead: 23.16 nsec
Histogram of timing durations:
< usec % of total count
1 97.70300 126549189
2 2.29506 2972668
4 0.00024 317
8 0.00008 101
16 0.00160 2072
32 0.00000 5
64 0.00000 6
128 0.00000 4
256 0.00000 0
512 0.00000 0
1024 0.00000 4
2048 0.00000 2
```

在我的例子中，在postgresql.conf文件中为一个会话打开track_io_timing的开销约为23纳秒，这很好。专业的高端服务器可以为你提供低至14纳秒的数字，而真正糟糕的虚拟化可以返回高达1400纳秒甚至1900纳秒的值。如果你使用的是云服务，你可以期待100-120纳秒左右（大多数情况下）。如果你曾经遇到过四位数的数值，测量I/O时序可能会导致真正可测量的开销，这将降低你的系统速度。一般的规则是这样的：在真实的硬件上，计时不是一个问题；在虚拟系统上，在你打开它之前要检查一下。

也可以通过使用ALTER DATABASE、ALTER USER等来有选择地打开一些东西。

1.1.3 检查表

一旦你对你的数据库中发生的事情有了大致的了解，那么深入挖掘并查看单个表中发生的情况可能是一个好主意。这里有两个系统视图可以帮助你：pg_stat_user_tables和pg_statio_user_tables。

这里是第一个。

```
test=# \d pg_stat_user_tables
View "pg_catalog.pg_stat_user_tables"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----+
relid | oid | |
schemaname | name | |
relname | name | |
seq_scan | bigint | |
seq_tup_read | bigint | |
idx_scan | bigint | |
idx_tup_fetch | bigint | |
n_tup_ins | bigint | |
n_tup_upd | bigint | |
n_tup_del | bigint | |
n_tup_hot_upd | bigint | |
n_live_tup | bigint | |
n_dead_tup | bigint | |
n_mod_since_analyze | bigint | |
last_vacuum | timestamp with time zone | |
last_autovacuum | timestamp with time zone | |
last_analyze | timestamp with time zone | |
last_autoanalyze | timestamp with time zone | |
vacuum_count | bigint | |
autovacuum_count | bigint | |
analyze_count | bigint | |
autoanalyze_count | bigint | |
```

根据我的判断，pg_stat_user_tables是最重要的系统视图之一，但也是最容易被误解甚至忽略的系统视图之一。我有一种感觉，许多人阅读了它，但却没有充分挖掘出这里真正可以看到的潜力。如果使用得当，pg_stat_user_tables在某些情况下，可以说是一种启示。

在我们深入研究数据的解释之前，重要的是要了解哪些字段是实际存在的。首先，每个表都有一个条目，它将显示发生在该表上的连续扫描的数量（seq_scan）。然后，我们有seq_tup_read，它告诉我们系统在这些顺序扫描中要读取多少个元组。

记住seq_tup_read列；它包含了可以帮助你找到性能问题的重要信息。

然后，idx_scan是名单上的下一个。它将向我们显示这个表使用索引的频率。PostgreSQL也会向我们显示这些扫描返回了多少行。然后，还有几列，以n_tup_开始。这些将告诉我们我们插入、更新和删除了多少数据。这里最重要的是与HOT UPDATE有关。当运行UPDATE时，PostgreSQL必须复制一条记录以确保ROLLBACK能正常工作。HOT UPDATE相当好，因为它允许PostgreSQL确保一行不需要离开一个区

块。

该行的副本保持在同一个区块内，这对一般的性能是有利的。相当数量的HOT UPDATE表明，在UPDATE工作量很大的情况下，你的方向是正确的。这里不能对所有的使用情况说明正常和HOT UPDATE之间的完美比例。你真的要自己想一想，找出哪种工作负载能从许多就地操作中受益。一般的规则是：UPDATE越密集的工作负载，越适合使用许多HOT UPDATE条款。

最后，还有一些VACUUM的统计数据，它们大多是不言自明的。

1.1.4 理解 pg_stat_user_tables

阅读所有这些数据可能很有趣；但是，除非你能够从中找出意义，否则它是非常没有意义的。使用pg_stat_user_tables的一个方法是检测哪些表可能需要一个索引。找出这个问题的一个方法是使用下面的查询。

```
SELECT schemaname, relname, seq_scan, seq_tup_read,
       seq_tup_read / seq_scan AS avg, idx_scan
  FROM pg_stat_user_tables
 WHERE seq_scan > 0
 ORDER BY seq_tup_read DESC LIMIT 25;
```

这个想法是为了找到在连续扫描中被频繁使用的大表。这些表会自然而然地出现在列表的顶部，为我们提供极高的seq_tup_read值，这可能是令人震惊的。

从上到下进行，寻找昂贵的扫描。请记住，顺序扫描不一定是坏事。它们自然而然地出现在备份、分析语句等方面，不会造成任何伤害。然而，如果你一直在运行大型的顺序扫描，你的性能将会下降。

请注意，这个查询是真正的黄金--它将帮助你发现缺少索引的表。我近二十年的实践经验一再表明，缺失索引是导致性能不佳的最重要原因。因此，你正在看的查询就像黄金一样。

一旦你完成了对可能缺失的索引的寻找，可以考虑简单看看你的表的缓存行为。为了方便起见，pg_statio_user_tables包含了各种信息，比如表的缓存行为（heap_blk），*你的索引的缓存行为*（idx_blk），以及The Oversized-Attribute Storage Technique (TOAST) 表的缓存行为。最后，你可以通过以下代码了解更多关于TID扫描（这只是按物理顺序读取数据的扫描），这通常与系统的整体性能无关。

```
test=# \d pg_statio_user_tables
View "pg_catalog.pg_statio_user_tables"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
relid | oid | |
schemaname | name | |
relname | name | |
heap_blk_read | bigint | |
heap_blk_hit | bigint | |
idx_blk_read | bigint | |
idx_blk_hit | bigint | |
toast_blk_read | bigint | |
toast_blk_hit | bigint | |
tidx_blk_read | bigint | |
tidx_blk_hit | bigint | |
```

尽管pg_statio_user_tables包含了重要的信息，但通常情况下，pg_stat_user_tables更有可能为你提供真正相关的见解（比如丢失索引等等）。

1.2 深入研究索引

虽然pg_stat_user_tables对于发现丢失的索引很重要，但有时也有必要找到那些不应该真正存在的索引。最近，我在德国出差，发现一个系统包含了大部分无意义的索引（占总存储消耗的74%）。虽然如果你的数据库真的很小，这可能不是一个问题，但在大系统的情况下，这确实是个问题--拥有数百GB的无意义的索引会严重损害你的整体性能。

幸运的是，可以通过检查pg_stat_user_indexes来找到那些无意义的索引。

```
test=# \d pg_stat_user_indexes
View "pg_catalog.pg_stat_user_indexes"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
relid | oid | || |
indexrelid | oid | || |
schemaname | name | || |
relname | name | || |
indexrelname | name | || |
idx_scan | bigint | || |
idx_tup_read | bigint | || |
idx_tup_fetch | bigint | || |
```

该视图告诉我们每个模式中每个表的每个索引被使用的频率 (idx_scan)。为了丰富这个视图，我建议使用以下SQL查询。

```
SELECT schemaname, relname, indexrelname, idx_scan,
pg_size.pretty(pg_relation_size(indexrelid)) AS idx_size,
pg_size.pretty(sum(pg_relation_size(indexrelid)))
OVER (ORDER BY idx_scan, indexrelid) AS total
FROM pg_stat_user_indexes
ORDER BY 6 ;
```

这个语句的输出是非常有用的。它不仅包含关于一个索引被使用的频率的信息--它还告诉我们每个索引浪费了多少空间。最后，它将所有的空间消耗加到了第6列中。现在你可以通过表格，重新思考所有那些很少被使用的索引。关于何时放弃一个索引，很难有一个普遍的规则，所以一些人工检查是很有意义的。

不要盲目地放弃索引。在某些情况下，索引根本没有被使用，因为终端用户使用应用程序的方式与预期的不同。如果最终用户发生了变化（例如，雇佣了一个新的秘书），一个索引很可能再次变成一个有用的对象。

还有一个叫做pg_statio_user_indexes的视图，它包含关于一个索引的缓存信息。虽然它很有趣，但它通常不包含导致大跳动的信息。

1.3 追踪后台工作者

在这一节中，我们将看一下后台写程序的统计数据。正如你可能已经知道的，数据库连接在许多情况下不会直接向磁盘写入块。相反，数据是由后台写程序或检查指针写入的。

要查看数据是如何写入的，请检查pg_stat_bgwriter视图。

```
test=# \d pg_stat_bgwriter
View "pg_catalog.pg_stat_bgwriter"
Column | Type | Collation | Nullable | Default
```

```

+-----+-----+-----+
|      |
|      |
| checkpoints_timed | bigint |   |
| checkpoints_req | bigint |   |
| checkpoint_write_time | double precision |   |
| checkpoint_sync_time | double precision |   |
| buffers_checkpoint | bigint |   |
| buffers_clean | bigint |   |
| maxwritten_clean | bigint |   |
| buffers_backend | bigint |   |
| buffers_backend_fsync | bigint |   |
| buffers_alloc | bigint |   |
| stats_reset | timestamp with time zone |   |

```

这里首先应该引起你的注意的是前两列。在本书的后面，你将了解到PostgreSQL会定期进行检查点，这对于确保数据真正进入磁盘是必要的。如果你的检查点之间距离太近，checkpoint_req可能会给你指出正确的方向。如果请求的检查点的数量很高，这可能意味着已经写了很多数据，而且由于高吞吐量，检查点总是被触发。除此之外，PostgreSQL还会告诉你在检查点期间写数据需要多少时间以及同步需要多少时间。另外，buffers_checkpoint表明在检查点期间有多少缓冲区被写入，有多少缓冲区被后台写手写入(buffers_clean)。

但还有更多：maxwritten_clean告诉我们后台写程序因为写了太多缓冲区而停止清理扫描的次数。

最后，还有buffers_backend（由后台数据库连接直接写入的缓冲区数量），buffers_backend_fsync（由数据库连接刷新的缓冲区数量），以及buffers_alloc（包含分配的缓冲区数量）。一般来说，如果数据库连接因为性能原因开始自己写东西，这不是一件好事。

1.4 跟踪、存档和流式传输

在本节中，我们将看一下与复制和交易日志存档有关的一些功能。首先要检查的是pg_stat_archiver，它告诉我们关于将事务日志（WAL）从主服务器转移到备份设备的存档过程。

```

test=# \d pg_stat_archiver
View "pg_catalog.pg_stat_archiver"
Column | Type | Collation | Nullable | Default
+-----+-----+-----+
archived_count | bigint |   |
last_archived_wal | text |   |
last_archived_time | timestamp with time zone |   |
failed_count | bigint |   |
last_failed_wal | text |   |
last_failed_time | timestamp with time zone |   |
stats_reset | timestamp with time zone |   |

```

此外，pg_stat_archiver包含关于你的归档过程的重要信息。首先，它将告知你已经归档的交易日志文件的数量（archived_count）。它还会知道最后一个被归档的文件和发生的时间（last_archived_wal 和 last_archived_time）。

虽然知道WAL文件的数量当然很有趣，但它其实并不那么重要。因此，可以考虑看一下 failed_count 和 last_failed_wal。如果你的事务日志归档失败了，它将告诉你最近一次失败的文件以及发生的时间。建议关注这些字段，因为，否则，有可能归档工作在你没有注意到的情况下进行。

如果你正在运行一个流式复制，下面两个视图对你来说非常重要。第一个被称为pg_stat_replication，它将提供关于流处理的信息。每个WAL发送器进程将有一个条目可见。如果没有一个条目，那么这意味着没有交易日志流在进行，这可能不是你想要的。

让我们看一下pg_stat_replication。

```
test=# \d pg_stat_replication
View "pg_catalog.pg_stat_replication"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----+
pid | integer | |||
usesysid | oid | |||
username | name | |||
application_name | text | |||
client_addr | inet | |||
client_hostname | text | |||
client_port | integer | |||
backend_start | timestamp with time zone | |||
backend_xmin | xid | |||
state | text | |||
sent_lsn | pg_lsn | |||
write_lsn | pg_lsn | |||
flush_lsn | pg_lsn | |||
replay_lsn | pg_lsn | |||
write_lag | interval | |||
flush_lag | interval | |||
replay_lag | interval | |||
sync_priority | integer | |||
sync_state | text | |||
reply_time | timestamp with time zone | |||
spill_txns | bigint | |||
spill_count | bigint | |||
spill_bytes | bigint | |||
```

在这里，你会发现一些列，表明通过流式复制连接的用户名。然后，还有应用程序名称，以及连接数据（client_）。在这里，PostgreSQL会告诉我们流式连接何时开始。在生产中，一个年轻的连接可能指向一个网络问题或更糟糕的事情（可靠性问题等等）。状态列显示了流的另一端处于何种状态。我们将在第10章“理解备份和复制”中更详细地介绍这个问题。

这里有一些字段告诉我们有多少事务日志已经通过网络连接发送（send_lsn），有多少已经发送到内核（write_lsn），有多少已经刷到磁盘（flush_lsn），还有多少已经被重放（replay_lsn）。最后，列出了同步状态。从PostgreSQL 10.0开始，还有一些额外的字段，已经包含了节点之间的时间差。lag字段包含了时间间隔，这在一定程度上表明了你的服务器之间的实际时间差。

PostgreSQL 13为这个视图增加了一些信息：spill字段告诉我们逻辑解码的行为方式。有时逻辑解码必须溢出到磁盘，这又会导致性能问题。通过检查spill_*字段，我们可以看到有多少事务（spill_txns）必须以多长时间（spill_count）和多少（spill_bytes）溢出到磁盘。虽然可以在复制设置的发送服务器上查询pg_stat_replication，但可以在接收端查询pg_stat_wal_receiver。它提供类似的信息并允许在副本上提取此信息。

下面是视图的定义：

```
test=# \d pg_stat_wal_receiver
View "pg_catalog.pg_stat_wal_receiver"
Column | Type | Collation | Nullable | Default
```

```

+-----+-----+-----+
| pid | integer | |
| status | text | |
| receive_start_lsn | pg_lsn | |
| receive_start_tli | integer | |
| written_lsn | pg_lsn | |
| flushed_lsn | pg_lsn | |
| received_tli | integer | |
| last_msg_send_time | timestamp with time zone | |
| last_msg_receipt_time | timestamp with time zone | |
| latest_end_lsn | pg_lsn | |
| latest_end_time | timestamp with time zone | |
| slot_name | text | |
| sender_host | text | |
| sender_port | integer | |
| conninfo | text | |

```

首先，PostgreSQL会告诉我们WAL接收器进程的进程ID。然后，视图向我们显示正在使用的连接状态。`receive_start_lsn`将告诉我们WAL接收器启动时使用的事务日志位置。除此之外，`receive_start_tli`还包含了WAL接收器启动时正在使用的时间线。在某些时候，你可能想知道最新的WAL位置和时间线。为了得到这两个数字，请使用`received_lsn`和`received_tli`。

在接下来的两列中，有两个时间戳：`last_msg_send_time`和`last_msg_receipt_time`。第一个说明了消息最后发送的时间，第二个说明了消息被接收的时间。

`latest_end_lsn`包含了在`latest_end_time`报告给WAL发送者进程的最后一个事务日志位置。然后，还有`slot_name`字段和连接信息的混淆版本。在PostgreSQL 11中，增加了额外的字段--`sender_host`、`sender_port`和`conninfo`字段告诉我们WAL接收器所连接的主机的情况。

1.5 检查 SSL 连接

许多运行PostgreSQL的用户使用SSL来加密从服务器到客户端的连接。最近版本的PostgreSQL提供了一个视图，这样我们就可以获得这些加密连接的概况，也就是`pg_stat_ssl`。

```

test=# \d pg_stat_ssl
  View "pg_catalog.pg_stat_ssl"
 Column | Type | Collation | Nullable | Default
+-----+-----+-----+
| pid | integer | |
| ssl | boolean | |
| version | text | |
| cipher | text | |
| bits | integer | |
| compression | boolean | |
| client_dn | text | |
| client_serial | numeric | |
| issuer_dn | text | |

```

每个进程都由进程ID代表。如果一个连接使用SSL，第二列被设置为真。第三和第四列将定义版本，以及密码。最后，还有加密算法使用的比特数，包括是否使用压缩的指标，以及来自客户端证书的区分名称（DN）字段。

1.6 实时检查事务情况

到目前为止，已经讨论了一些统计表。所有这些表背后的想法都是为了查看整个系统中发生的事情。但是，如果你是一个想检查单个事务的开发者，该怎么办呢？`pg_stat_xact_user_tables`在这里提供了帮助。它不包含整个系统的事务，它只包含关于你当前事务的数据。这在下面的代码中显示。

```
test=# \d pg_stat_xact_user_tables
View "pg_catalog.pg_stat_xact_user_tables"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
relid | oid | |
schemaname | name | |
relname | name | |
seq_scan | bigint | |
seq_tup_read | bigint | |
idx_scan | bigint | |
idx_tup_fetch | bigint | |
n_tup_ins | bigint | |
n_tup_upd | bigint | |
n_tup_del | bigint | |
n_tup_hot_upd | bigint | |
```

`pg_stat_xact_user_tables`的内容与你在`pg_stat_user_tables`中看到的内容基本相同。但是，背景有些不同。

开发人员可以在一个事务提交之前查看它是否造成了任何性能问题。这可以帮助我们区分整体数据和刚刚由我们的应用程序引起的数据。

应用程序开发人员使用这个视图的理想方式是在提交之前在应用程序中添加一个函数调用，以跟踪事务所做的事情。

然后可以检查这个数据，这样就可以将当前事务的输出与整个工作负载区分开来。

1.7 跟踪 VACUUM 和 CREATE INDEX 进度

在PostgreSQL 9.6中，社区引入了一个许多人期待已久的系统视图。许多年来，人们一直想跟踪真空进程的进度，看看事情可能需要多长时间。

由于这个原因，发明了`pg_stat_progress_vacuum`来解决这个问题。下面的列表显示了你可以获得什么样的信息。

```
test=# \d pg_stat_progress_vacuum
View "pg_catalog.pg_stat_progress_vacuum"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
pid | integer | |
datid | oid | |
datname | name | |
relid | oid | |
phase | text | |
heap_blk_total | bigint | |
heap_blk_scanned | bigint | |
heap_blk_vacuumed | bigint | |
index_vacuum_count | bigint | |
max_dead_tuples | bigint | |
num_dead_tuples | bigint | |
```

大多数栏目不言自明，因此我不会在此过多地讨论细节。只是有几件事情应该牢记在心。首先，这个过程不是线性的--它可以有相当大的跳跃。除此以外，真空通常是相当快的，所以进展可能很快，而且很难跟踪。从PostgreSQL 12开始，也有一种方法可以看到CREATE INDEX正在做什么。

pg_stat_progress_create_index是与pg_progress_vacuum相对应的。下面是系统视图的定义。

```
test=# \d pg_stat_progress_create_index
  View "pg_catalog.pg_stat_progress_create_index"
 Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
 pid | integer | |
 datid | oid | |
 datname | name | |
 relid | oid | |
 index_relid | oid | |
 command | |
 phase | text | |
 lockers_total | bigint | |
 lockers_done | bigint | |
 current_locker_pid | bigint | |
 blocks_total | bigint | |
 blocks_done | bigint | |
 tuples_total | bigint | |
 tuples_done | bigint | |
 partitions_total | bigint | |
 partitions_done | bigint | |
```

这个表的内容有助于我们了解CREATE INDEX进行到什么程度。为了向你展示这个表的内容是什么样子的，我创建了一个相当大的可以被索引的表。

```
test=# CREATE TABLE t_index (x numeric);
CREATE TABLE
test=# INSERT INTO t_index
    SELECT * FROM generate_series(1, 50000000);
INSERT 0 50000000
test=# CREATE INDEX idx_numeric ON t_index (x);
CREATE INDEX
```

在创建索引的过程中，有很多阶段。首先，PostgreSQL要扫描你想建立索引的表，在系统视图中表示如下。

```
test=# SELECT * FROM pg_stat_progress_create_index;
-[ RECORD 1 ]-----+
pid | 805
datid | 16546
datname | test
relid | 24576
index_relid | 0
command | CREATE INDEX
phase | building index: scanning table
lockers_total | 0
lockers_done | 0
current_locker_pid | 0
blocks_total | 221239
```

```
blocks_done | 59872
tuples_total | 0
tuples_done | 0
partitions_total | 0
partitions_done | 0
```

一旦这样做了，PostgreSQL就会实际建立真正的索引，这也可以在系统表里面看到，如下面的代码清单所示。

```
test=# SELECT * FROM pg_stat_progress_create_index;
-[ RECORD 1 ]
-----+
pid | 29191
datid | 16410
datname | test
relid | 24600
index_relid | 0
command | CREATE INDEX
phase | building index: loading tuples in tree
lockers_total | 0
lockers_done | 0
current_locker_pid | 0
blocks_total | 0
blocks_done | 0
tuples_total | 50000000
tuples_done | 4289774
partitions_total | 0
partitions_done | 0
```

在我的列表中，接近10%的加载过程已经完成，如列表最后的tuples_*列所示。

1.8 使用 pg_stat_statements

现在我们已经讨论了前几个视图，现在是时候把注意力转向最重要的一个视图了，它 can 以用来发现性能问题。当然，我指的是pg_stat_statements。这个想法是为了掌握你系统中的查询信息。它可以帮助我们弄清楚哪些类型的查询速度慢，以及查询被调用的频率如何。

要使用这个模块，我们需要遵循三个步骤。

1. 在postgresql.conf文件中把pg_stat_statements添加到shared_preload_libraries中。
2. 重新启动数据库服务器。
3. 在你选择的数据库中运行CREATE EXTENSION pg_stat_statements

让我们检查一下视图的定义。

```
test=# \d pg_stat_statements
View "public.pg_stat_statements"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----+
userid | oid | |||
dbid | oid | |||
queryid | bigint | |||
query | text | |||
plans | bigint | |||
total_plan_time | double precision | |||
```

```

min_plan_time | double precision | |
max_plan_time | double precision | |
mean_plan_time | double precision | |
stddev_plan_time | double precision | |
calls | bigint | |
total_exec_time | double precision | |
min_exec_time | double precision | |
max_exec_time | double precision | |
mean_exec_time | double precision | |
stddev_exec_time | double precision | |
rows | bigint | |
shared_blk_hit | bigint | |
shared_blk_read | bigint | |
shared_blk_dirtied | bigint | |
shared_blk_written | bigint | |
local_blk_hit | bigint | |
local_blk_read | bigint | |
local_blk_dirtied | bigint | |
local_blk_written | bigint | |
temp_blk_read | bigint | |
temp_blk_written | bigint | |
blk_read_time | double precision | |
blk_write_time | double precision | |
wal_records | bigint | |
wal_fpi | bigint | |
wal_bytes | numeric | |

```

有趣的是，`pg_stat_statements`提供的信息简直令人难以置信。对于每个数据库中的每个用户，它为每个查询提供一行。默认情况下，它跟踪5000条语句（这可以通过设置`pg_stat_statements.max`来改变）。

查询和参数是分开的。PostgreSQL会把占位符放入查询中。这允许相同的查询，只是使用不同的参数，被聚合起来。例如，`SELECT ... FROM x WHERE y = 10`将变成`SELECT ... FROM x WHERE y = ?`

对于每个查询，PostgreSQL会告诉我们它所消耗的总时间，以及它所做的调用次数。在最近的版本中，增加了`min_time`、`max_time`、`mean_time`和`stddev`。标准偏差尤其值得注意，因为它将告诉我们一个查询的运行时间是稳定的还是波动的。不稳定的运行时间可能由于各种原因而发生。

- 如果数据没有完全缓存在RAM中，需要到磁盘上的查询将比缓存的查询花费更多的时间。
- 不同的参数会导致不同的计划和完全不同的结果集。
- 并发和锁定也会产生影响。

在PostgreSQL 13中的新内容是，系统也会告诉我们一些关于优化器性能和计划时间的信息。如果计划是一个问题，这就非常重要。对于小的查询，计划可能是相当昂贵的（与整个运行时间相比）。

PostgreSQL也会告诉我们一个查询的缓存行为。共享列告诉我们有多少块来自缓存（hit）或来自操作系统（_read）。如果许多块来自操作系统，查询的运行时间可能会有波动。

下一个列块是关于本地缓冲区的。本地缓冲区是由数据库连接直接分配的内存块。

在所有这些信息之上，PostgreSQL提供了关于临时文件I/O的信息。请注意，当建立一个大型索引或执行其他一些大型DDL时，临时文件I/O会自然发生。然而，在OLTP中，临时文件通常是一个非常糟糕的事情，因为它们会通过潜在的磁盘阻塞来减慢整个系统的速度。大量的临时文件I/O可以指出一些不理想的事情。下面的列表包含了三种常见的情况。

- 不理想的`work_mem`设置（OLTP）
- 不理想的`maintenance_work_mem`设置（DDL）
- 一开始就不应该运行的查询

最后，有两个字段包含有关I/O计时的信息。默认情况下，这两个字段是空的。原因是，在一些系统上，测量时间可能涉及到相当多的开销。因此，track_io_timing的默认值是false，如果你需要这些数据，记得打开它。

在PostgreSQL 13中还有一个新的内容是关于WAL生成的信息。你可以发现已经产生了多少WAL（记录数和字节数）。这可以让你对数据库的性能有宝贵的了解。

一旦该模块被启用，PostgreSQL就会收集数据，你可以使用该视图。

永远不要在客户面前运行SELECT * FROM pg_stat_statements。众所周知，人们会开始指着他们碰巧知道的查询，并让他们解释为什么，谁，什么，什么时候，等等。当你使用这个视图时，总是创建一个排序的输出，这样就可以立即看到最相关的信息。

下面的查询可以证明对获得数据库服务器上发生的事情的概述非常有帮助。如果不知道发生了什么，调试几乎是不可能的。pg_stat_statements是一个非常好的方法，可以找出哪些是慢的，哪些是不慢的。

```
test=# SELECT round((100 * total_exec_time / sum(total_exec_time)
OVER ()::numeric, 2) percent,
round(total_exec_time::numeric, 2) AS total,
calls,
round(mean_exec_time::numeric, 2) AS mean,
substring(query, 1, 40)
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 10;
percent | total | calls | mean | substring
-----+-----+-----+-----+
-
-
66.27 | 319648.78 | 55859 | 5.72 | UPDATE pgbench_accounts SET abalance = a
18.54 | 89423.28 | 1 | 89423.28 | copy pgbench_accounts from stdin
6.37 | 30729.70 | 1 | 30729.70 | vacuum analyze pgbench_accounts
6.20 | 29886.45 | 1 | 29886.45 | alter table pgbench_accounts add primary
1.92 | 9270.97 | 55859 | 0.17 | UPDATE pgbench_branches SET bbalance = b
0.37 | 1770.88 | 55859 | 0.03 | UPDATE pgbench_tellers SET tbalance = tb
0.13 | 608.56 | 55859 | 0.01 | SELECT abalance FROM pgbench_accounts WH
0.10 | 493.33 | 55859 | 0.01 | INSERT INTO pgbench_history (tid, bid, a
0.05 | 239.06 | 1 | 239.06 | vacuum analyze pgbench_branches
0.02 | 112.91 | 1 | 112.91 | vacuum analyze pgbench_tellers
(10 rows)
```

前面的代码显示了前10个查询和它们的运行时间，包括一个百分比。显示查询的平均执行时间也是有意义的，这样你就可以决定这些查询的运行时间是否过高。

沿着列表往下走，检查所有似乎平均运行时间过长的查询。

请记住，对前1000个查询进行处理通常是不值得的。在大多数情况下，前几个查询已经承担了系统中的大部分负载。

在我的例子中，我使用了一个子串来缩短查询的时间，这样就可以在一页纸上完成。如果你真的想知道发生了什么，我不建议这样做。

请记住，pg_stat_statements默认会在1024字节时切断查询。下面的配置可以控制这种行为。

```
test=# SHOW track_activity_query_size;
track_activity_query_size
```

```
-----  
1024  
(1 row)
```

可以考虑把这个值增加到，比如说，16,384。如果你的客户正在运行基于Hibernate的Java应用程序，一个更大的track_activity_query_size的值将确保查询不会在有趣的部分显示之前被切断。

在这一点上，我想用这种情况来指出pg_stat_statements到底有多重要。到目前为止，它是追踪性能问题的最简单的方法。慢速查询日志永远不会像pg_stat_statements那样有用，因为慢速查询日志只会指向个别的慢速查询--它不会向我们展示由大量的中等查询引起的问题。因此，建议总是打开这个模块。其开销真的很小，而且丝毫不损害系统的整体性能。

默认情况下，有5000种查询被跟踪。在大多数合理合理的应用中，这就足够了。要重置数据，可以考虑使用以下指令。

```
test=# SELECT pg_stat_statements_reset();
```

```
pg_stat_statements_reset
```

```
-----  
(1 row)
```

偶尔重设一下统计数字会有很大的意义，以确保你能看到最新的信息，而不是一些历史的旧信息。

2 创建日志文件

现在我们已经深入了解了PostgreSQL提供的系统视图，现在是配置日志的时候了。幸运的是，PostgreSQL为我们提供了一种简单的方法来处理日志文件，并帮助人们轻松设置好配置。

收集日志很重要，因为它可以指出错误和潜在的数据库问题。在本节中，你将学习如何正确配置日志。

postgresql.conf文件包含了所有你需要的参数，这样就为你提供了所有必要的信息。

2.1 配置 postgresql.conf 文件

在本节中，我们将浏览postgresql.conf文件中一些最重要的条目，我们可以用这些条目来配置日志，并看看如何以最有利的方式使用日志。

在我们开始之前，我想就PostgreSQL中的日志记录说几句，一般来说。在Unix系统中，PostgreSQL默认会将日志信息发送到stderr。然而，stderr并不是一个好的地方，因为你会在某些时候想要检查日志流。因此，你应该通过本章的学习，根据你的需要来调整事情。让我们看一下，看看如何做到这一点。

2.2 定义日志的目的地和轮换

配置日志很容易，但需要一点知识。让我们浏览一下postgresql.conf文件，看看可以做什么。

```
#-
# REPORTING AND LOGGING
#-
# - Where to Log -
#log_destination = 'stderr'
# valid values are combinations of
```

```
# stderr, csvlog, syslog, and eventlog,  
# depending on platform. csvlog  
# requires logging_collector to be on.  
# This is used when logging to stderr:  
logging_collector = off  
# Enable capturing of stderr and csvlog  
# into log files. Required to be on for  
# csvlogs.  
# (change requires restart)
```

第一个配置选项定义了日志的处理方式。默认情况下，它将转到stderr（在Unix）。在Windows上，默认是eventlog，它是Windows的板载工具，用来处理日志。另外，你可以选择使用csvlog或syslog。

如果你想制作PostgreSQL的日志文件，你应该选择stderr并打开日志收集器。然后PostgreSQL将创建日志文件。

现在的逻辑问题是：这些日志文件的名称将是什么，这些文件将存放在哪里？幸运的是，`postgresql.conf`给出了答案。

```
# These are only used if logging_collector is on:  
#log_directory = 'pg_log'  
# directory where log files are written,  
# can be absolute or relative to PGDATA  
#log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'  
# log file name pattern,  
# can include strftime() escapes
```

`log_directory`将告诉系统在哪里存储日志。如果你使用的是绝对路径，你可以明确地配置日志的去向。如果你希望日志直接在PostgreSQL的数据中，只需选择一个相对路径。这样做的好处是，数据目录将是自成一体的，你可以不必担心移动它。

在下一步，你可以定义PostgreSQL应该使用的文件名。PostgreSQL非常灵活，允许你使用strftime提供的所有快捷方式。为了让你知道这个功能有多强大，在我的平台上快速统计了一下，发现strftime提供了43个（！）占位符来创建文件名。用户通常需要的一切当然都是可能的。

一旦文件名被定义，简单地考虑一下清理问题是有意义的。以下设置将是可用的

```
#log_truncate_on_rotation = off  
#log_rotation_age = 1d  
#log_rotation_size = 10MB
```

默认情况下，如果文件超过1天或大于10MB，PostgreSQL将继续生产日志文件。

此外，`log_truncate_on_rotation`指定你是否要向日志文件追加数据。有时，`log_filenames`的定义方式会让它变成循环的。`log_truncate_on_rotation`参数定义了是覆盖还是追加到已经存在的文件中。鉴于默认的日志文件，这当然不会发生。一种处理自动轮换的方法是使用诸如`postgresql_%a.log`，以及`log_truncate_on_rotation = on`。`%a`的意思是在日志文件中使用一周的日期。这里的好处是，一周中的每一天每7天重复一次。因此，日志将被保存一个星期并被回收。如果你的目标是每周轮换，10MB的文件大小可能是不够的。考虑把最大文件大小关掉。

2.3 配置系统日志

有些人喜欢使用syslog来收集日志文件。PostgreSQL提供以下配置参数。

```
# These are relevant when logging to syslog:  
#syslog_facility = 'LOCAL0'  
#syslog_ident = 'postgres'  
#syslog_sequence_numbers = on  
#syslog_split_messages = on
```

syslog在系统管理员中相当流行。幸运的是，它很容易配置。基本上，你设置一个设施和一个标识符。如果log_destination被设置为syslog，那么你就不需要做其他事情了。

2.4 记录慢速查询

日志也可以用来追踪个别缓慢的查询。在过去，这几乎是发现性能问题的唯一方法。

它是如何工作的呢？基本上，postgresql.conf有一个叫做log_min_duration_statement的变量。如果这个变量被设置为大于0，每一个超过我们所选择的设置的查询都会被记录到日志中。

```
# log_min_duration_statement = -1
```

大多数人将慢速查询日志视为智慧的最终来源。然而，我想补充一点警告的话。有很多慢速查询，而它们恰好占用了大量的CPU：索引创建、数据导出、分析，等等。

这些长时间运行的查询是完全可以预期的，而且在很多情况下不是万恶之源。经常出现的情况是，许多较短的查询都是罪魁祸首。这里有一个例子：1,000次查询x 500毫秒比2次查询x 5秒更糟糕。在某些情况下，缓慢的查询日志可能会产生误导。

尽管如此，这并不意味着它毫无意义--它只是意味着它是一个信息来源，而不是信息的来源。

2.5 定义记录内容和方式

在看了一些基本设置后，现在是时候决定记录什么了。默认情况下，只有错误会被记录下来。然而，这可能是不够的。在本节中，你将了解什么可以被记录，以及日志的内容是什么。

默认情况下，PostgreSQL不会记录关于检查点的信息。下面的设置正是为了改变这一点。

```
#log_checkpoints = off
```

这同样适用于连接；每当一个连接被建立或适当地销毁时，PostgreSQL可以创建日志条目。

```
#log_connections = off  
#log_disconnections = off
```

在大多数情况下，记录连接是没有意义的，因为大量的记录会大大降低系统的速度。分析系统不会受到太大的影响。但是，OLTP可能会受到严重影响。如果你想看看语句需要多长时间，可以考虑把下面的设置切换到开。

```
#log_duration = off
```

让我们继续讨论最重要的一个设置。我们还没有定义信息的布局，到目前为止，日志文件包含以下形式的错误。

```
test=# SELECT 1/0;  
psql: ERROR: division by zero
```

日志中会注明ERROR，同时还有错误信息。在PostgreSQL 10.0之前，并没有时间戳、用户名等内容。你必须立即改变这个值，才能对日志有任何意义。在PostgreSQL 10.0中，默认值已经变成了更合理的东西。要改变这一点，请看一下log_line_prefix。

```
#log_line_prefix = '%m [%p] '
# special values:
# %a = application name
# %u = user name
# %d = database name
# %r = remote host and port
# %h = remote host
# %p = process ID
# %t = timestamp without milliseconds
# %m = timestamp with milliseconds
# %n = timestamp with milliseconds (as a Unix epoch)
# %i = command tag
# %e = SQL state
# %c = session ID
# %l = session line number
# %s = session start timestamp
# %v = virtual transaction ID
# %x = transaction ID (0 if none)
# %q = stop here in non-session processes
# %% = '%'
```

log_line_prefix是相当灵活的，允许你配置logline以完全满足你的需要。一般来说，记录一个时间戳是个好主意。否则，几乎不可能看到什么时候发生了不好的事情。就个人而言，我也喜欢知道用户名、事务ID和数据库。然而，这取决于你决定你真正需要什么。

有时，速度慢是由不好的锁定行为造成的。用户之间的相互阻塞会导致糟糕的性能，理清这些问题以确保高吞吐量是很重要的。一般来说，与锁有关的问题可能很难追踪。

基本上，log_lock_waits可以帮助检测此类问题。如果一个锁被保持的时间超过了deadlock_timeout，那么就会有一行被发送到日志中，前提是以下配置变量被打开。

```
#log_lock_waits = off
```

最后，是时候告诉PostgreSQL实际要记录什么了。到目前为止，只有错误、缓慢的查询，以及类似的情况被发送到日志中。然而，log_statement有四种可能的设置，如下面这块所示。

```
#log_statement = 'none'
# none, ddl, mod, all
```

注意，none意味着只有错误会被记录。ddl意味着错误以及DDL（CREATE TABLE, ALTER TABLE, 等等）都会被记录。mod已经包括了数据变化，all会把每条语句都发送到日志中。

请注意，所有这些都会导致大量的日志信息，这可能会减慢你的系统。

```
#log_replication_commands = off
```

这将与复制相关的命令发送到日志。

关于复制的更多信息，请访问以下网站：<https://www.postgresql.org/docs/current/static/protocol-replication.html>。

经常会有这样的情况：性能问题是由临时文件I/O引起的。要查看哪些查询会导致问题，可以使用以下设置。

```
#log_temp_files = -1
# log temporary files equal or larger
# than the specified size in kilobytes;
# -1 disables, 0 logs all temp files
```

pg_stat_statements包含综合信息，而log_temp_files将指向导致问题的具体查询。通常把这个设置成一个合理的低值是有意义的。正确的值取决于你的工作负载，但也许4MB是一个好的开始。

默认情况下，PostgreSQL会在服务器所在的时区写入日志文件。然而，如果你正在运行一个分布在世界各地的系统，调整时区的方式是有意义的，这样你就可以去比较日志条目，如下代码所示。

```
log_timezone = 'Europe/Vienna'
```

请记住，在SQL方面，你仍然会看到你本地时区的时间。然而，如果设置了这个变量，日志条目将在不同的时区。如果打开日志记录，会导致巨大的日志量，这反过来又会降低数据库的性能。已经添加了三个参数来控制产生的日志数量。

```
log_min_duration_sample = -1
log_statement_sample_rate = 1.0
log_transaction_sample_rate = 0.0
```

采样率定义了这些所需的日志条目中，有多少会实际进入日志流中。这个想法是为了减少日志的数量，同时仍然保持这些设置的有用性。

日志是查看数据库中正在发生的事情的关键。然而，我们建议明智地配置日志，以避免产生过多的日志，从而导致速度减慢。根据需要提取尽可能多的信息，但要避免过度。

3 总结

这一章是关于系统统计的。你学会了如何从PostgreSQL中提取信息，以及如何以有益的方式使用系统统计。没有适当的监控，实现良好和可靠的运行几乎是不可能的。密切关注运行时参数和数据库的生命体征以避免麻烦是很重要的。本章已经教会了你很多关于PostgreSQL监控的知识，这些知识可以用来优化你的数据库。

对最重要的视图进行了详细的讨论。第6章，优化查询以获得良好的性能，这是本书的下一章，都是关于查询优化。你将学习如何检查查询，以及如何对其进行优化。

4 问题

就像本书的大多数章节一样，我们将看一下从刚才的内容中产生的一些关键问题。

- PostgreSQL收集什么样的运行时统计数据？
- 我怎样才能容易地发现性能问题？
- PostgreSQL是如何写日志文件的？
- 日志对性能有影响吗？

这些问题的答案可以在GitHub仓库中找到 (<https://github.com/PacktPublishing/Mastering-PostgreSQL-13-Fourth-Edition>) 。

优化查询以获得良好的性能

1 学习优化器的作用

- 1.1 一个实际的例子--查询优化器如何处理一个样本查询
- 1.2 评估连接方案
 - 1.2.1 嵌套循环 Nested loops
 - 1.2.2 哈希连接 Hash joins
 - 1.2.3 合并连接 Merge joins
- 1.3 应用转换
- 1.4 应用等式约束
- 1.5 详尽的搜索
- 1.6 检查执行计划
- 1.7 使过程失败
- 1.8 不断折叠
- 1.9 理解函数内联
- 1.9 引入连接修剪
- 1.10 加快集合操作

2 了解执行计划

- 2.1 系统地接近计划
- 2.2 使 EXPLAIN 更冗长
- 2.3 发现问题
 - 2.3.1 发现运行时的变化
 - 2.3.2 检查估算
 - 2.3.3 检查缓冲区使用情况
 - 2.3.4 修复缓冲区的高使用率

3 了解和修复连接

- 3.1 获得左连接
- 3.2 处理外连接
- 3.3 了解 joinCollapse_limit 变量

4 启用和禁用优化器设置

- 4.1 了解遗传查询优化

5 分区数据

- 5.1 创建继承表
- 5.2 应用表约束
- 5.3 修改继承结构
- 5.4 将表移入和移出分区结构
- 5.5 清理数据
- 5.6 了解PostgreSQL 13.x的分区功能

6 调整参数以获得良好的查询性能

- 6.1 加快排序
- 6.2 加快管理任务

7 利用并行查询

- 7.1 PostgreSQL 能够并行做什么?
- 7.2 实践中的并行性

8.引入即时编译 (JIT)

- 8.1 配置 JIT
- 8.2 运行查询

9 总结

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重要的，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。让我们假设这三个表，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 $1\text{亿} \times 2\text{亿}$ 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 $O(n^2)$ ，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
-----
10.000000000000000
(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_1d (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_1d 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_1d 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 $0.2 \times 0.2 =$ 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 $5,000 \times 5\text{毫秒} = 25\text{秒}$ 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 b 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
-----+
 2.000000000000000 | 2.000000000000000
(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段，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/hansjurgenscho_nig/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查看这些东西。

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发现在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.cyberite.com/postgresql/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。

编写存储过程

1 理解存储过程语言

- 1.1 了解存储过程与函数的基本原理
- 1.2 函数的剖析
- 1.3 介绍美元引号
- 1.4 使用匿名代码块
- 1.5 使用函数和事务

2 探索各种存储过程语言

- 2.1 介绍 PL/pgSQL
 - 2.1.1 处理引号和字符串格式
 - 2.1.2 管理作用域
 - 2.1.3 了解高级错误处理
 - 2.1.4 使用 GET DIAGNOSTICS
 - 2.1.5 使用游标来获取分块的数据
 - 2.1.6 使用复合类型
 - 2.1.7 在 PL/pgSQL 中编写触发器
 - 2.1.8 在 PL/pgSQL 中编写存储过程
- 2.2 介绍PL/Perl
 - 2.2.1 利用PL/Perl进行数据类型抽象
 - 2.2.2 在 PL/Perl 和 PL/PerlU 之间做出选择
 - 2.2.3 使用 SPI 接口
 - 2.2.4 使用 SPI 设置返回函数
 - 2.2.5 在 PL/Perl 中转义和支持函数
 - 2.2.6 跨函数调用共享数据
 - 2.2.7 在 Perl 中编写触发器
- 2.3 介绍 PL/Python
 - 2.3.1 编写简单的 PL/Python 代码
 - 2.3.2 使用 SPI 接口
 - 2.3.3 处理错误

3 改进函数

- 3.1 减少函数调用次数
- 3.2 使用缓存计划
- 3.3 将成本分配给函数

4 为各种目的使用函数

5 总结

6 问题

在第6章 "优化查询以获得良好性能 "中，我们了解了很多关于优化器以及系统中正在进行的优化。本章将介绍存储过程，以及如何有效和方便地使用它们。你将了解到存储过程是由什么组成的，哪些语言可以使用，以及你如何能够很好地加速事情。除此之外，你还会被介绍到PL/pgSQL的一些更高级的功能。本章将涵盖以下主题。

- 理解存储过程语言
- 探索各种存储过程语言
- 改进函数
- 为各种目的使用函数

在本章结束时，你将能够编写良好、高效的程序

1 理解存储过程语言

当涉及到存储过程和函数时，PostgreSQL与其他数据库系统有相当大的不同。大多数数据库引擎强迫你使用某种编程语言来编写服务器端代码。Microsoft SQL Server提供Transact-SQL，而Oracle鼓励你使用PL/SQL。PostgreSQL不强迫你使用某种语言；相反，它允许你决定你最熟悉和喜欢的语言。

PostgreSQL之所以如此灵活，从历史意义上讲，其实也很有意思。许多年前，最著名的PostgreSQL开发者之一，Jan Wieck，在PostgreSQL的早期就写了无数的补丁，提出了使用工具命令语言（Tcl）作为服务器端编程语言的想法。问题是，没有人愿意使用Tcl，也没有人愿意在数据库引擎中使用这些东西。解决这个问题的办法是使语言界面非常灵活，基本上任何语言都可以很容易地与PostgreSQL集成。在这一点上，CREATE LANGUAGE子句诞生了。下面是CREATE LANGUAGE的语法。

```
test=# \h CREATE LANGUAGE
Command: CREATE LANGUAGE
Description: define a new procedural language
Syntax:
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR valfunction ]
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
URL: https://www.postgresql.org/docs/13/sql-createlanguage.html
```

现在，许多不同的语言都可以用来编写函数和存储过程。为PostgreSQL增加的灵活性确实得到了回报；我们现在可以从丰富的编程语言中选择。

PostgreSQL到底是如何处理语言的？如果我们看一下CREATE LANGUAGE子句的语法，我们会看到一些关键字。

- HANDLER：这个函数实际上是PostgreSQL和你想使用的任何外部语言之间的粘合剂。它负责将PostgreSQL的数据结构映射到语言所需要的东西上，并帮助传递代码。
- VALIDATOR：这是基础设施的警察。如果它是可用的，它将负责向终端用户提供可口的语法错误。许多语言能够在实际执行代码之前解析它。PostgreSQL可以利用这一点，在你创建一个函数的时候告诉你这个函数是否正确。不幸的是，不是所有的语言都能做到这一点，所以在某些情况下，你仍然会在运行时出现问题。
- INLINE：如果存在这个，PostgreSQL将能够利用这个处理函数来运行匿名代码块。

1.1 了解存储过程与函数的基本原理

在我们深入了解存储过程的结构之前，有必要先谈谈一般的函数和过程。传统上，存储过程这一术语实际上是用来谈论一个函数的。因此，我们必须了解函数和存储过程之间的区别。

一个函数是普通SQL语句的一部分，不允许启动或提交事务。下面是一个例子。

```
SELECT func(id) FROM large_table;
```

假设func(id)被调用了5000万次。如果你使用名为Commit的函数，到底应该发生什么？不可能简单地在查询过程中结束一个事务并启动一个新的事务。整个交易的完整性、一致性等概念都会被违反。

相反，一个存储过程能够控制事务，甚至能够一个接一个地运行多个事务。但是，你不能在一个SELECT语句中运行它。相反，你必须调用CALL。下面的列表显示了CALL命令的语法。

```
test=# \h CALL
Command: CALL
Description: invoke a procedure
Syntax:
CALL name ( [ argument ] [ , ... ] )
URL: https://www.postgresql.org/docs/13/sql-call.html
```

因此，功能和程序之间有一个基本的区别。你在互联网上找到的术语并不总是很清楚。然而，你必须意识到这些重要的区别。在PostgreSQL中，函数从一开始就已经存在了。然而，本节所概述的过程的概念是新的，在PostgreSQL 11中才被引入。在本章中，我们将详细介绍函数和过程。让我们从了解函数的结构开始。

1.2 函数的剖析

在我们研究具体的语言之前，们将看一下典型函数的剖析。出于演示的目的，让我们看看下面这个函数，它只是将两个数字相加。

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int)
RETURNS int AS
'
SELECT $1 + $2;
' LANGUAGE 'sql';
CREATE FUNCTION
```

首先要注意的是，这个函数是用SQL写的。PostgreSQL需要知道我们使用的是哪种语言，所以我们必须在定义中指定。

请注意，函数的代码是以字符串(')的形式传递给PostgreSQL的。这是值得注意的，因为它允许一个函数成为一个执行机器的黑匣子。

在其他数据库引擎中，函数的代码不是一个字符串，而是直接连接到语句中。这个简单的抽象层给了PostgreSQL函数管理器所有的力量。在字符串里面，你基本上可以使用你所选择的编程语言所能提供的一切。

在这个例子中，我们将简单地把传递给函数的两个数字相加。有两个整数变量在使用。这里重要的部分是，PostgreSQL为你提供了函数重载。换句话说，mysum(int, int)函数与mysum(int8, int8)函数是不一样的。

PostgreSQL认为这些东西是两个不同的函数。函数重载是一个很好的功能；但是，如果你的参数列表恰好时常变化，你必须非常小心，不要意外地部署太多的函数。一定要确保不再需要的函数真的被删除。

CREATE OR REPLACE FUNCTION子句不会改变参数列表。因此，只有在签名不改变的情况下，你才能使用它。它要么会出错，要么会简单地部署一个新函数。让我们运行mysum函数。

```
test=# SELECT mysum(10, 20);
mysum
-----
30
(1 row)
```

这里的結果是30，这其实并不令人惊讶。在介绍完这些函数之后，重要的是要关注下一个主要话题：引用。

1.3 介绍美元引号

将代码作为字符串传递给PostgreSQL是非常灵活的。然而，使用单引号可能是一个问题。在许多编程语言中，单引号经常出现。为了能够使用这些引号，人们必须在把字符串传递给PostgreSQL时转义它们。许多年来，这一直是标准程序。幸运的是，那些旧时代已经过去了，现在有新的手段可以将代码传递给PostgreSQL。其中之一是美元引号，如下面的代码所示。

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int)
RETURNS int AS
$$
SELECT $1 + $2;
$$ LANGUAGE 'sql';
CREATE FUNCTION
```

你可以不使用引号来开始和结束字符串，而直接使用\$\$. 目前，有两种语言已经为\$\$赋予了意义。在Perl以及Bash脚本中，\$代表进程ID。为了克服这个小障碍，我们几乎可以在任何东西前面使用\$来开始和结束字符串。下面的例子说明了这一方法的作用。

```
test=# CREATE OR REPLACE FUNCTION mysum(int, int) RETURNS int AS
$body$
SELECT $1 + $2;
$body$ LANGUAGE 'sql';
CREATE FUNCTION
```

所有这些灵活性使你能够一劳永逸地克服引号的问题。只要开始字符串和结束字符串相匹配，就完全不会有任何问题。

1.4 使用匿名代码块

到目前为止，我们已经编写了尽可能简单的存储过程，也学会了如何执行代码。然而，代码的执行不仅仅是完整的函数。除了函数之外，PostgreSQL还允许使用匿名代码块。这个想法是为了运行只需要一次的代码。这种代码执行在处理管理任务时特别有用。匿名代码块不接受参数，也不会永久地保存在数据库中，因为它们没有名字。

下面是一个简单的例子，显示了一个匿名代码块的运行情况。

```
test=# DO
$$
BEGIN
RAISE NOTICE 'current time: %', now();
END;
$$ LANGUAGE 'plpgsql';
NOTICE: current time: 2020-09-01 09:52:39.279219+02
CONTEXT: PL/pgSQL function inline_code_block line 3 at RAISE
DO
```

在这个例子中，代码只发出了一条信息，然后就退出了。同样，代码块必须知道它使用哪种语言。这个字符串被传递给PostgreSQL，使用简单的美元引号

1.5 使用函数和事务

如你所知，PostgreSQL在用户区暴露的所有东西都是一个事务。当然，如果你正在编写函数，也是如此。一个函数总是你所处的事务的一部分。它不是独立的，就像一个操作者或任何其他操作。

这是一个例子：

```
test=# SELECT now(), mysum(id, id) FROM generate_series(1, 3) AS id;
now | mysum
-----
2020-09-01 09:52:55.966338+02 | 2
2020-09-01 09:52:55.966338+02 | 4
2020-09-01 09:52:55.966338+02 | 6
(3 rows)
```

所有三个函数调用都发生在同一个事务中。这一点很重要，因为它意味着你不能在一个函数内做太多的事务性流程控制。当第二个函数调用提交时会发生什么？它就是不能工作。

然而，Oracle有一种机制，允许自主交易。这个想法是，即使一个事务回滚，有些部分可能仍然需要，它们应该被保留。一个经典的例子是这样的。

1. 启动一个查找秘密数据的功能。
2. 在文件中添加一个日志行，说明有人修改了这个重要的秘密数据。
3. 提交日志行，但回滚修改。
4. 保存信息，说明有人试图改变数据。

为了解决类似这样的问题，可以使用自主事务。这个想法是为了能够在主事务中独立提交一个事务。在这种情况下，日志表中的条目将占上风，而变化将被回滚。

从PostgreSQL 11.0开始，自主事务并没有实现。然而，已经有一些补丁在流传，以实现这一功能。这些功能什么时候能进入核心版本还有待观察。

为了给你一个印象，让你知道事情很可能是如何运作的，这里有一个基于第一批补丁的代码片段。

```
...
AS
$$
DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    FOR i IN 0..9 LOOP
        START TRANSACTION;
        INSERT INTO test1 VALUES (i);
        IF i % 2 = 0 THEN
            COMMIT;
        ELSE
            ROLLBACK;
        END IF;
    END LOOP;
    RETURN 42;
END;
$$;
...
```

这个例子的重点是告诉你，我们可以决定是提交还是回滚自主事务。

2 探索各种存储过程语言

正如我们在本章中已经说过的，PostgreSQL让你有能力用各种语言编写函数和存储过程。以下是可用的选项，并与PostgreSQL核心一起提供。

- SQL PL/pgSQL
- PL/Perl and PL/PerlU
- PL/Python
- PL/Tcl and PL/TclU

SQL是编写函数的明显选择，应该尽可能地使用它，因为它给了优化器最大的自由。然而，如果你想写稍微复杂一点的代码，PL/pgSQL可能是你的选择语言。

PL/pgSQL提供了流程控制和更多的功能。在这一章中，我们将展示PL/pgSQL的一些更高级和不为人知的特性，但请记住，这一章并不意味着是PL/pgSQL的完整教程。

核心部分包含在Perl中运行服务器端函数的代码。基本上，这里的逻辑是一样的。代码将以字符串的形式传递并由Perl执行。请记住，PostgreSQL不会说Perl，它只是有代码将东西传给外部编程语言。

也许你已经注意到Perl和Tcl有两种风格：可信的（PL/Perl和PL/Tcl）和不可信的（PL/PerlU和PL/TclU）。受信任和不受信任的语言之间的区别实际上是一个重要的区别。在PostgreSQL中，一种语言被直接加载到数据库连接中。因此，该语言能够做相当多的讨厌的事情。为了摆脱安全问题，人们发明了可信语言的概念。这个想法是，可信语言被限制在语言的核心部分。它不可能做以下事情。

- 包括库
- 打开网络套接字
- 执行任何类型的系统调用，这将包括打开文件

Perl提供了一种叫做污点模式的东西，它被用来在PostgreSQL中实现这一功能。Perl会自动将自己限制在信任模式下，如果即将发生安全漏洞，就会出错。在非信任模式下，一切皆有可能，因此只有超级用户可以运行非信任的代码。

如果你想运行受信任以及不受信任的代码，你必须激活两种语言，即plperl和plperlu（分别是pltcl和pltclu）。

目前，Python只能作为一种不受信任的语言使用；因此，当涉及到一般的安全问题时，管理员必须非常小心，因为在不受信任模式下运行的函数可以绕过所有由PostgreSQL执行的安全机制。只要记住，Python是作为你的数据库连接的一部分来运行的，而绝不是对安全负责。

让我们开始讨论本章中最令人期待的话题。

2.1 介绍 PL/pgSQL

在这一节中，你将被介绍到PL/pgSQL的一些更高级的特性，这些特性对于编写正确和高效的代码非常重要。请注意，这不是对编程的初学者介绍，也不是对PL/pgSQL的一般介绍。

2.1.1 处理引号和字符串格式

数据库编程中最重要的事情之一是引号。如果你不使用正确的引号，你肯定会陷入SQL注入的麻烦中，并打开不可接受的安全漏洞。

什么是SQL注入？让我们考虑下面的例子。

```
CREATE FUNCTION broken(text) RETURNS void AS
$$
DECLARE
    v_sql text;
BEGIN
    v_sql := 'SELECT schemaname
    FROM pg_tables
    WHERE tablename = ''' || $1 || '''';
    RAISE NOTICE 'v_sql: %', v_sql;
    RETURN;
END;
$$ LANGUAGE 'plpgsql';
```

在这个例子中，SQL代码被简单地粘贴在一起，而不必担心安全问题。我们在这里所做的就是使用`||`操作符来连接字符串。如果人们运行正常的查询，这就很好。考虑一下下面的例子，显示了一些破损的代码。

```
SELECT broken('t_test');
```

然而，我们必须准备好应对那些试图利用你的系统的人。考虑一下下面的例子。

```
SELECT broken('''; DROP TABLE t_test; ');
```

带着这个参数运行函数会显示一个问题。下面的代码显示了经典的SQL注入。

```
NOTICE: v_sql: SELECT schemaname FROM pg_tables
WHERE tablename = ''; DROP TABLE t_test; '
CONTEXT: PL/pgSQL function broken(text) Line 6 at RAISE
broken
-----
(1 row)
```

当你只想做一个查询时，删除一个表并不是一件理想的事情。让你的应用程序的安全取决于传递给你的语句的参数，这绝对是不可接受的。为了避免SQL注入，PostgreSQL提供了各种函数；这些函数应该随时使用，以确保你的安全保持不变。

```
test=# SELECT quote_literal(E'o''reilly'), quote_ident(E'o''reilly');
quote_literal | quote_ident
-----+-----
'o''reilly' | "o'reilly" (1 row)
```

`quote_literal`函数将以这样的方式转义一个字符串，即不会再发生任何坏事。它将在字符串周围加上所有的引号，并转义字符串中有问题的字符。因此，不需要手动开始和结束字符串。这里显示的第二个函数是`quote_ident`。它可以用来正确引用对象名称。注意，使用的是双引号，这正是处理表名所需要的。下面的例子显示了如何使用复杂的名称。

```
test=# CREATE TABLE "Some stupid name" ("ID" int);
CREATE TABLE
test=# \d "Some stupid name"
Table "public.Some stupid name"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
 ID | integer | | |
```

通常情况下，PostgreSQL中所有的表名都是小写的。然而，如果使用了双引号，对象名称可以包含大写字母。一般来说，使用这种技巧并不是一个好主意，因为你将不得不一直使用双引号，这可能有点不方便。

现在你已经有了一个关于引号的基本介绍，重要的是看一下NULL值是如何处理的。下面的代码显示了quote_literal函数是如何处理NULL的。

```
test=# SELECT quote_literal(NULL);
quote_literal
-----
(1 row)
```

如果你在一个NULL值上调用quote_literal函数，它将直接返回NULL。在这种情况下，没有必要照顾到引号。PostgreSQL提供了更多的函数来明确地处理NULL值。

```
test=# SELECT quote_nullable(123), quote_nullable(NULL);
quote_nullable | quote_nullable
-----+-----+
'123' | NULL (1 row)
```

它不仅可以引用字符串和对象名称；还可以使用PL/pgSQL上的格式化和准备整个查询。这里的美妙之处在于，你可以使用格式化函数来向语句中添加参数。下面是它的工作原理。

```
CREATE FUNCTION simple_format() RETURNS text AS
$$
DECLARE
v_string text;
v_result text;
BEGIN
v_string := format('SELECT schemaname|| ''.'' || tablename
FROM pg_tables
WHERE %I = $1
AND %I = $2', 'schemaname', 'tablename');
EXECUTE v_string USING 'public', 't_test' INTO v_result;
RAISE NOTICE 'result: %', v_result;
RETURN v_string;
END;
$$ LANGUAGE 'plpgsql';
```

字段的名称被传递给format函数。最后，EXECUTE语句的USING子句是为了将参数添加到查询中，然后执行。同样，这里的好处是不可能发生SQL注入。

下面是调用simple_format函数时发生的情况。

```
test=# SELECT simple_format ();
NOTICE: result: public .t_test
simple_format
-----
SELECT schemaname|| '.' || tablename +
FROM pg_tables +
WHERE schemaname = $1+
AND tablename = $2
(1 row)
```

正如你所看到的，调试信息正确地显示了表，包括模式，并正确地返回了查询。然而，格式函数可以做得更多。下面是一些例子。

```
test=# SELECT format('Hello, %s %s','PostgreSQL', 13);
format
-----
Hello, PostgreSQL 13
(1 row)
test=# SELECT format('Hello, %s %10s','PostgreSQL', 13);
format
-----
Hello, PostgreSQL 13
(1 row)
```

format能够使用格式选项，如例子所示。%10s意味着我们要添加的字符串将被填充。空白会被添加。

在某些情况下，可能需要不止一次地使用一个变量。下面的例子显示了两个参数，它们被不止一次地添加到我们想要创建的字符串中。你可以做的是用1美元、2美元等来识别参数列表中的条目。

```
test=# SELECT format('%1$s, %1$s, %2$s', 'one', 'two');
format
-----
one, one, two
(1 row)
```

format是一个非常强大的功能，当你想避免SQL注入时，这个功能超级重要，你应该好好利用这个强大的功能。

2.1.2 管理作用域

在处理了一般的引用和基本安全（SQL注入）之后，我们将关注另一个重要的话题：作用域。就像大多数流行的编程语言一样，PL/pgSQL使用变量，这取决于其上下文。变量是在一个函数的DECLARE语句中定义的。然而，PL/pgSQL允许你嵌套一个DECLARE语句。

```
CREATE FUNCTION scope_test () RETURNS int AS
$$
DECLARE
    i int := 0;
BEGIN
    RAISE NOTICE 'i1: %', i;
    DECLARE
        i int;
    BEGIN
        RAISE NOTICE 'i2: %', i;
```

```
END;
RETURN i;
END;
$$ LANGUAGE 'plpgsql';
```

在DECLARE语句中，定义了i变量并给它分配了一个值。然后，i被显示出来。然后，第二个DECLARE语句开始。它包含了一个额外的i的化身，它没有被分配一个值。因此，其值将是NULL。注意，PostgreSQL现在将显示内部的i。

```
test=# SELECT scope_test();
NOTICE: i1: 0
NOTICE: i2: <NULL>
scope_test
-----
0
(1 row)
```

正如所料，调试信息将显示0和NULL。PostgreSQL允许你使用各种各样的技巧。然而，我们强烈建议你保持你的代码简单和易读。

2.1.3 了解高级错误处理

对于编程语言，在每个程序和每个模块中，错误处理是一件重要的事情。每件事情都会偶尔出错，因此，正确和专业地处理错误是至关重要的，也是关键性的。在PL/pgSQL中，你可以使用EXCEPTION块来处理错误。我们的想法是，如果BEGIN块做错了什么，EXCEPTION块会处理它，并正确地处理问题。就像许多其他语言，如Java，你可以对不同类型的错误做出反应，并分别捕捉它们。

在下面的例子中，代码可能会遇到一个除以零的问题。我们的目标是抓住这个错误并作出相应的反应。

```
CREATE FUNCTION error_test1(int, int) RETURNS int AS
$$
BEGIN
RAISE NOTICE 'debug message: % / %', $1, $2;
BEGIN
RETURN $1 / $2;
EXCEPTION
WHEN division_by_zero THEN
RAISE NOTICE 'division by zero detected: %', sqlerrm;
WHEN others THEN
RAISE NOTICE 'some other error: %', sqlerrm;
END;
RAISE NOTICE 'all errors handled';
RETURN 0;
END;
$$ LANGUAGE 'plpgsql';
```

BEGIN块显然可以抛出一个错误，因为有可能出现除以0的情况。然而，EXCEPTION块抓住了我们正在看的错误，同时也处理了所有其他可能意外出现的潜在问题。

从技术上讲，这或多或少与保存点相同，因此，错误不会导致整个事务完全失败。只有导致错误的区块才会受到小型回滚的影响。

通过检查sqlerrm变量，你也可以直接接触到错误信息本身。让我们运行这段代码。

```
test=# SELECT error_test1(9, 0);
NOTICE: debug message: 9 / 0
NOTICE: division by zero detected: division by zero
NOTICE: all errors handled
error_test1
-----
0
(1 row)
```

PostgreSQL捕获了这个异常，并在EXCEPTION块中显示了这个消息。它还很友好的向我们展示了出错的那一行。这使得调试和修复代码变得更加容易，如果它被破坏的话。

在某些情况下，引发你自己的异常也是有意义的。正如你所期望的，这是很容易做到的。

```
RAISE uniqueViolation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

除此之外，PostgreSQL提供了许多预定义的错误代码和异常。下面的页面包含这些错误信息的完整列表：<https://www.postgresql.org/docs/13/static/errcodes-appendix.html>。

2.1.4 使用 GET DIAGNOSTICS

许多过去使用过Oracle的人可能对GET DIAGNOSTICS子句很熟悉。GET DIAGNOSTICS子句背后的想法是让用户看到系统中正在发生的事情。虽然对于习惯于现代代码的人来说，这种语法可能显得有些奇怪，但它仍然是一个有价值的工具，可以使你的应用程序变得更好。

从我的观点来看，GET DIAGNOSTICS子句有两个主要任务可以使用。

- 检查行数
- 获取上下文信息并获得回溯

检查行数绝对是你在日常编程中需要的。如果你想调试应用程序，提取上下文信息将很有用。

下面的例子显示了如何在你的代码中使用GET DIAGNOSTICS子句。

```
CREATE FUNCTION get_diag() RETURNS int AS
$$
DECLARE
    rc int;
    _sqlstate text;
    _message text;
    _context text;
BEGIN
    EXECUTE 'SELECT * FROM generate_series(1, 10)';
    GET DIAGNOSTICS rc = ROW_COUNT;
    RAISE NOTICE 'row count: %', rc;
    SELECT rc / 0;
EXCEPTION
    WHEN OTHERS THEN
        GET STACKED DIAGNOSTICS
            _sqlstate = returned_sqlstate,
            _message = message_text,
            _context = pg_exception_context;
        RAISE NOTICE 'sqlstate: %, message: %, context: [%]',
            _sqlstate,
            _message,
            replace(_context, E'\n', ' <- ');
```

```
    RETURN rc;
END;
$$ LANGUAGE 'plpgsql';
```

声明这些变量后的第一件事是执行一个SQL语句，并向GET DIAGNOSTICS子句询问行数，然后在调试信息中显示。然后，该函数强制PL/pgSQL出错。一旦发生这种情况，我们将使用GET DIAGNOSTICS子句从服务器上提取信息来显示。

下面是我们调用get_diag函数时的情况。

```
test=# SELECT get_diag();
NOTICE: row count: 10
CONTEXT: PL/pgSQL function get_diag() line 12 at RAISE
NOTICE: sqlstate: 22012,
message: division by zero,
context: [SQL statement "SELECT rc / 0"
<- PL/pgSQL function get_diag() line 14 at
SQL statement]
CONTEXT: PL/pgSQL function get_diag() line 22 at RAISE
get_diag
-----
10
(1 row)
```

正如你所看到的，GET DIAGNOSTICS子句给了我们关于系统中活动的详细信息。

2.1.5 使用游标来获取分块的数据

如果你执行SQL，数据库将计算出结果并将其发送给你的应用程序。一旦整个结果集被发送至客户端，应用程序就可以继续做它的工作。问题是这样的：如果结果集大到无法再装入内存，会发生什么？如果数据库返回100亿行怎么办？客户端应用程序通常不能一次处理这么多数据，实际上，它也不应该这样做。解决这个问题的方法是游标。游标背后的想法是，只有在需要时（调用FETCH时）才会产生数据。因此，在数据库实际生成数据的时候，应用程序已经可以开始消耗数据了。除此之外，执行这一操作所需的内存要少得多。

当涉及到PL/pgSQL时，游标也起到了重要作用。每当你在一个结果集上循环时，PostgreSQL会在内部自动使用游标。这样做的好处是，你的应用程序的内存消耗将大大减少，而且由于处理的数据量很大，几乎不可能出现内存耗尽的情况。使用游标的方式有多种。

下面是一个最简单的例子，在一个函数内使用游标。

```
CREATE OR REPLACE FUNCTION c(int)
RETURNS setof text AS
$$
DECLARE
v_rec record;
BEGIN
FOR v_rec IN SELECT tablename
FROM pg_tables
LIMIT $1
LOOP
RETURN NEXT v_rec.tablename;
END LOOP;
RETURN;
END;
```

```
$$ LANGUAGE 'plpgsql';
```

这段代码很有趣，有两个原因。首先，它是一个集合返回函数（SRF）。它产生了整个列，而不仅仅是一个单行。实现这一目的的方法是使用变量集，而不仅仅是数据类型。RETURN NEXT子句将建立起结果集，直到我们到达终点。RETURN子句将告诉PostgreSQL我们要离开这个函数，并且我们有了结果。

第二个重要的问题是，在游标上循环将自动创建一个内部游标。换句话说，不需要担心有可能耗尽内存。PostgreSQL将对查询进行优化，试图尽可能快地产生前10%的数据（由cursor_tuple_fraction变量定义）。下面是这个查询将返回的内容。

```
test=# SELECT * FROM c(3);
c
-----
t_test
pg_statistic
pg_type
(3 rows)
```

在这个例子中，只是会有一个随机表的列表。如果你那边的结果不一样，这在一定程度上是可以预期的。

在我看来，你刚才看到的是在PL/pgSQL中使用隐式游标的最频繁和最常见的方式。

下面的例子显示了一个较老的机制，许多有Oracle背景的人可能知道这个机制。

```
CREATE OR REPLACE FUNCTION d(int)
RETURNS setof text AS
$$
DECLARE
v_cur refcursor;
v_data text;
BEGIN
OPEN v_cur FOR
SELECT tablename
FROM pg_tables
LIMIT $1;
WHILE true LOOP
FETCH v_cur INTO v_data;
IF FOUND THEN
RETURN NEXT v_data;
ELSE
RETURN;
END IF;
END LOOP;
END;
$$ LANGUAGE 'plpgsql';
```

在这个例子中，游标被明确地声明和打开。在里面，循环数据被显式地获取并返回给调用者。基本上，查询做的是同样的事情。这只是一个关于开发人员实际喜欢什么语法的品味问题。

你还觉得你对游标的了解还不够吗？还有，这里有第三个选项，可以做完全相同的事情。

```
CREATE OR REPLACE FUNCTION e(int)
RETURNS setof text AS
$$
DECLARE
```

```
v_cur CURSOR (param1 int) FOR
SELECT tablename
FROM pg_tables
LIMIT param1;
v_data text;
BEGIN
OPEN v_cur ($1);
WHILE true LOOP
FETCH v_cur INTO v_data;
IF FOUND THEN
RETURN NEXT v_data;
ELSE
RETURN;
END IF;
END LOOP;
END;
$$ LANGUAGE 'plpgsql';
```

在这种情况下，游标被输入一个整数参数，这个参数直接来自于函数调用（\$1）。有时，游标并没有被存储过程本身使用，而是被返回供以后使用。在这种情况下，你可以返回一个简单的游标作为返回值。

```
CREATE OR REPLACE FUNCTION cursor_test(c refcursor)
RETURNS refcursor AS
$$
BEGIN
OPEN c FOR SELECT *
FROM generate_series(1, 10) AS id;
RETURN c;
END;
$$ LANGUAGE plpgsql;
```

这里的逻辑很简单。游标的名称被传递给函数。然后，游标被打开并返回。这里的好处是，游标背后的查询可以在运行中创建，并动态地编译。

应用程序可以像其他应用程序一样从游标中获取信息。下面是它的工作原理。

```
test=# BEGIN;
BEGIN
test=# SELECT cursor_test('mytest');
cursor_test
-----
mytest
(1 row)
test=# FETCH NEXT FROM mytest;
id
-----
1
(1 row)
test=# FETCH NEXT FROM mytest;
id
-----
2
(1 row)
```

请注意，它只在使用事务块时起作用。

在本节中，我们已经了解到游标只会在数据被消耗时产生数据。这对大多数查询来说是正确的。然而，这个例子有一个问题；只要使用SRF，整个结果就必须被物化。它不是即时创建的，而是一次性的。其原因是SQL必须能够重新扫描一个关系，这在普通表的情况下很容易做到。然而，对于函数，情况是不同的。因此，一个SRF总是被计算和物化，使得这个例子中的游标完全没有用。换句话说，我们在编写函数时需要小心。在某些情况下，危险就隐藏在巧妙的细节中。

2.1.6 使用复合类型

在大多数其他数据库系统中，存储过程只用于原始数据类型，比如整数、数字、varchar等等。然而，PostgreSQL是非常不同的。我们可以使用所有可用的数据类型。这包括原始的、复合的和自定义的数据类型。就数据类型而言，根本没有任何限制。为了充分释放PostgreSQL的力量，复合类型是非常重要的，并且经常被在互联网上发现的扩展所使用。下面的例子显示了如何将复合类型传递给一个函数，以及如何在内部使用它。最后，复合类型将被再次返回。

```
CREATE TYPE my_cool_type AS (s text, t text);
CREATE FUNCTION f(my_cool_type)
RETURNS my_cool_type AS
$$
DECLARE
v_row my_cool_type;
BEGIN
RAISE NOTICE 'schema: (%) / table: (%)'
, $1.s, $1.t;
SELECT schemaname, tablename
INTO v_row
FROM pg_tables
WHERE tablename = trim($1.t)
AND schemaname = trim($1.s)
LIMIT 1;
RETURN v_row;
END;
$$ LANGUAGE 'plpgsql';
```

这里的主要问题是，你可以简单地使用`1.field_name`来访问复合类型。返回复合类型也不难。

你只需要像其他数据类型一样，临时组装复合类型的变量并返回它。你甚至可以轻松地使用数组，甚至更复杂的结构。

下面的代码显示了PostgreSQL将返回什么。

```
test=# SELECT (f).s, (f).t
  FROM f ('("public", "t_test")'::my_cool_type);
NOTICE: schema: (public) / table: ( t_test)
 s | t
-----
 public | t_test
(1 row)
```

2.1.7 在 PL/pgSQL 中编写触发器

如果你想对数据库中发生的某些事件作出反应，服务器端代码就特别受欢迎。触发器允许你在一个表发生INSERT、UPDATE、DELETE或TRUNCATE子句时调用一个函数。被触发器调用的函数可以修改表中发生变化的数据，或者简单地执行一个必要的操作。

在PostgreSQL中，触发器多年来变得更加强大，它们现在提供了一系列丰富的功能。

```

test=# \h CREATE TRIGGER
Command: CREATE TRIGGER
Description: define a new trigger
Syntax:
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR
... ] }
[ ON table_name
[ FROM referenced_table_name ]
[ NOT DEFERRABLE | DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ]
]
[ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ]
]
[ FOR [ EACH ] { ROW | STATEMENT } ]
[ WHEN ( condition ) ]
EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )
where event can be one of:
INSERT
UPDATE [ OF column_name [, ... ] ]
DELETE
TRUNCATE
URL: https://www.postgresql.org/docs/13/sql-createtrigger.html

```

首先要注意的是，触发器总是为一个表或一个视图触发，并调用一个函数。它有一个名字，可以在一个事件之前或之后发生。PostgreSQL的魅力在于，你可以在一个表中有无数个触发器。虽然这对PostgreSQL的铁杆用户来说并不奇怪，但我想指出，这在许多昂贵的商业数据库引擎中是不可能的，这些引擎目前仍在世界各地使用。如果同一个表有多个触发器，那么多年前在PostgreSQL 7.3中引入的以下规则将是有用的：触发器是按字母顺序触发。首先，所有这些BEFORE触发器都是按字母顺序发生的。然后，PostgreSQL执行触发器被触发的行操作，并继续按字母顺序在触发器之后执行。换句话说，触发器的执行顺序是绝对确定的，而且触发器的数量基本上是无限的。

触发器可以在实际修改发生之前或之后修改数据。一般来说，这是一个验证数据的好方法，如果违反了一些自定义限制，就会出错。下面的例子显示了一个在INSERT子句中被触发的触发器，它改变了被添加到表中的数据。

```

CREATE TABLE t_sensor (
    id serial,
    ts timestamp,
    temperature numeric
);

```

我们的表只是存储了几个值。现在的目标是一旦有行插入，就调用一个函数。

```

CREATE OR REPLACE FUNCTION trig_func()
RETURNS trigger AS
$$
BEGIN
IF NEW.temperature < -273
THEN
NEW.temperature := 0;
END IF;
RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';

```

正如我们之前所说，触发器将总是调用一个函数，这允许你使用漂亮的抽象代码。这里重要的一点是，触发器函数必须返回一个触发器。要访问你要插入的行，你可以访问NEW变量。

INSERT和UPDATE触发器总是提供一个NEW变量。UPDATE和DELETE会提供一个叫做OLD的变量。这些变量包含你将要修改的行。

在我的例子中，代码检查温度是否过低。如果是的话，这个值是不行的，它会被动态地调整。为了确保修改后的行能够被使用，仅仅返回NEW。如果在这个触发器之后还有第二个触发器被调用，那么下一个函数调用将已经看到修改过的行。

在下一步，可以使用CREATE TRIGGER命令来创建触发器。

```
CREATE TRIGGER sensor_trig
BEFORE INSERT ON t_sensor
FOR EACH ROW
EXECUTE PROCEDURE trig_func();
```

这是触发器将执行的操作：

```
test=# INSERT INTO t_sensor (ts, temperature)
VALUES ('2017-05-04 14:43', -300) RETURNING *;
 id | ts | temperature
-----+----+-----
 1 | 2017-05-04 14:43:00 | 0
(1 row)

INSERT 0 1
```

正如你所看到的，该值已被正确调整。表中的内容显示温度为0。

如果你正在使用触发器，你应该意识到这样一个事实，即一个触发器对自己了解很多。它可以访问一些变量，这些变量允许你编写更复杂的代码，从而实现更好的抽象性。

让我们先删除触发器。

```
test=# DROP TRIGGER sensor_trig ON t_sensor;
DROP TRIGGER
```

触发器被成功删除。然后，可以添加一个新的函数。

```
CREATE OR REPLACE FUNCTION trig_demo()
RETURNS trigger AS
$$
BEGIN
RAISE NOTICE 'TG_NAME: %', TG_NAME;
RAISE NOTICE 'TG_RELNAME: %', TG_RELNAME;
RAISE NOTICE 'TG_TABLE_SCHEMA: %', TG_TABLE_SCHEMA;
RAISE NOTICE 'TG_TABLE_NAME: %', TG_TABLE_NAME;
RAISE NOTICE 'TG_WHEN: %', TG_WHEN;
RAISE NOTICE 'TG_LEVEL: %', TG_LEVEL;
RAISE NOTICE 'TG_OP: %', TG_OP;
RAISE NOTICE 'TG_NARGS: %', TG_NARGS;
-- RAISE NOTICE 'TG_ARGV: %', TG_NAME;
RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER sensor_trig
BEFORE INSERT ON t_sensor
FOR EACH ROW
EXECUTE PROCEDURE trig_demo();
```

这里使用的所有变量都是预定义的，默认是可用的。我们的代码所做的就是显示它们，以便我们可以看到它们的内容。

```
test=# INSERT INTO t_sensor (ts, temperature)
VALUES ('2017-05-04 14:43', -300) RETURNING *;
NOTICE: TG_NAME: demo_trigger
NOTICE: TG_RELNAME: t_sensor
NOTICE: TG_TABLE_SCHEMA: public
NOTICE: TG_TABLE_NAME: t_sensor
NOTICE: TG_WHEN: BEFORE
NOTICE: TG_LEVEL: ROW
NOTICE: TG_OP: INSERT
NOTICE: TG_NARGS: 0
 id | ts | temperature
-----+----+-----
 2 | 2017-05-04 14:43:00 | -300
(1 row)

INSERT 0 1
```

我们在这里看到的是，触发器知道它的名字，它被触发的表，以及很多其他的东西。为了在不同的表中应用类似的操作，这些变量有助于避免重复的代码，只需编写一个函数。然后，这可以用于我们感兴趣的所有表。

到目前为止，我们已经看到了简单的行级触发器，它在每个语句中被触发一次。然而，随着PostgreSQL 10.0的推出，有一些新的功能。语句级的触发器已经存在了一段时间了。然而，它不可能访问被触发器改变的数据。在PostgreSQL 10.0中，这个问题已经得到了解决，现在可以利用过渡表，其中包含所有的变化。

下面的代码包含一个完整的例子，显示了如何使用过渡表。

```
CREATE OR REPLACE FUNCTION transition_trigger()
RETURNS TRIGGER AS $$

DECLARE
v_record record;

BEGIN
IF (TG_OP = 'INSERT') THEN
RAISE NOTICE 'new data: ';
FOR v_record IN SELECT * FROM new_table
LOOP
RAISE NOTICE '%', v_record;
END LOOP;
ELSE
RAISE NOTICE 'old data: ';
FOR v_record IN SELECT * FROM old_table
LOOP
RAISE NOTICE '%', v_record;
END LOOP;
END IF;
RETURN NULL; -- result is ignored since this is an AFTER trigger
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER transition_test_trigger_ins
AFTER INSERT ON t_sensor
REFERENCING NEW TABLE AS new_table
FOR EACH STATEMENT EXECUTE PROCEDURE transition_trigger();
CREATE TRIGGER transition_test_trigger_del
AFTER DELETE ON t_sensor
REFERENCING OLD TABLE AS old_table
FOR EACH STATEMENT EXECUTE PROCEDURE transition_trigger();
```

在这种情况下，我们需要两个触发器定义，因为我们不能把所有东西都挤到一个定义里。在触发器函数内部，过渡表很容易使用：它可以像普通表一样被访问。

让我们通过插入一些数据来测试一下触发器的代码。

```
INSERT INTO t_sensor
SELECT *, now(), random() * 20
FROM generate_series(1, 5);
DELETE FROM t_sensor;
```

在我的例子中，代码将简单地为过渡表中的每个条目发出NOTICE。

```
NOTICE: new data:
NOTICE: (1,"2017-10-04 15:47:14.129151",10.4552665632218)
NOTICE: (2,"2017-10-04 15:47:14.129151",12.8670312650502)
NOTICE: (3,"2017-10-04 15:47:14.129151",14.3934494629502)
NOTICE: (4,"2017-10-04 15:47:14.129151",4.35718866065145)
NOTICE: (5,"2017-10-04 15:47:14.129151",10.9121138229966)
INSERT 0 5
NOTICE: old data:
NOTICE: (1,"2017-10-04 15:47:14.129151",10.4552665632218)
NOTICE: (2,"2017-10-04 15:47:14.129151",12.8670312650502)
NOTICE: (3,"2017-10-04 15:47:14.129151",14.3934494629502)
NOTICE: (4,"2017-10-04 15:47:14.129151",4.35718866065145)
NOTICE: (5,"2017-10-04 15:47:14.129151",10.9121138229966)
DELETE 5
```

请记住，使用数十亿行的过渡表不一定是个好主意。PostgreSQL确实是可扩展的，但在某些时候，有必要看到对性能也有影响。

2.1.8 在 PL/pgSQL 中编写存储过程

现在，让我们继续前进，学习如何编写存储过程。在本节中，你将学习如何编写真正的存储过程，这是PostgreSQL 11中引入的。要创建一个存储过程，你必须使用CREATE PROCEDURE。这个命令的语法与CREATE FUNCTION非常相似。只是有一些细微的差别，可以从下面的语法定义中看到。

```
test=# \h CREATE PROCEDURE
Command: CREATE PROCEDURE
Description: define a new procedure
Syntax:
CREATE [ OR REPLACE ] PROCEDURE
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [,
... ] ] )
    { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ] }
```

```
| [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER  
| SET configuration_parameter { TO value | = value | FROM CURRENT }  
| AS 'definition'  
| AS 'obj_file', 'link_symbol'  
} ...  
URL: https://www.postgresql.org/docs/13/sql-createprocedure.html
```

下面的例子显示了一个运行两个事务的存储过程。第一个事务是COMMIT，因此创建了两个表。第二个存储过程是ROLLBACK。

```
test=# CREATE PROCEDURE test_proc()  
LANGUAGE plpgsql  
AS $$  
BEGIN  
CREATE TABLE a (aid int); CREATE TABLE b (bid int);  
COMMIT;  
CREATE TABLE c (cid int);  
ROLLBACK;  
END;  
$$;  
CREATE PROCEDURE
```

正如我们所看到的，存储过程能够进行明确的事务处理。存储过程背后的想法是能够运行批处理工作和其他操作，这在函数中很难做到。

为了运行存储过程，你必须使用CALL，如下面的例子所示。

```
test=# CALL test_proc();  
CALL
```

前两个表已经提交由于过程中的回滚，第三个表尚未创建

```
test=# \d  
List of relations  
 Schema | Name | Type | Owner  
-----+-----+-----+  
 public | a | table | hs  
 public | b | table | hs  
(2 rows)
```

程序是PostgreSQL 11中引入的最重要的功能之一，它们对软件开发的效率做出了重大贡献。

2.2 介绍PL/Perl

关于PL/pgSQL，还有很多要讲的。然而，在一本书中不可能涵盖所有的内容，所以现在是时候转向下一个过程性语言了。PL/Perl已经被很多人采纳，成为字符串处理的理想语言。正如你可能知道的那样，Perl以其字符串操作能力而闻名，因此这么多年后仍然相当流行。

要启用PL/Perl，你有两个选择。

```
test=# create extension plperl;  
CREATE EXTENSION  
test=# create extension plperlu;  
CREATE EXTENSION
```

你可以部署受信任或不受信任的Perl。如果你想要这两种语言，你必须同时启用这两种语言。

为了向你展示PL/Perl是如何工作的，我实现了一个简单解析电子邮件地址并返回真或假的函数。下面是它的原理。

```
test=# CREATE OR REPLACE FUNCTION verify_email(text)
RETURNS boolean AS
$$
if ($_[0] =~ /^[a-zA-Z0-9.]+@[a-zA-Z0-9.-]+\$/)
{
    return true;
}
return false;
$$ LANGUAGE 'plperl';
CREATE FUNCTION
```

一个测试参数被传递给函数。在函数内部，所有这些输入参数都可以用\$_访问。在这个例子中，正则表达式被执行，而函数被返回。

该函数可以被调用，就像用其他语言编写的其他过程一样。下面的列表显示了如何调用该函数。

```
test=# SELECT verify_email('hs@cybertec.at');
verify_email
-----
t
(1 row)
test=# SELECT verify_email('totally wrong');
verify_email
-----
f
(1 row)
```

前面的列表显示，该函数正确验证了代码。请记住，如果你在一个受信任的函数内部，你就不能加载包，等等。例如，如果你想用w命令来查找单词，Perl会在内部加载utf8.pm，当然，这是不允许的。

2.2.1 利用PL/Perl进行数据类型抽象

正如本章已经说过的，PostgreSQL中的函数是相当通用的，可以在许多不同的情况下使用。如果你想使用函数来提高你的数据质量，你可以使用CREATE DOMAIN子句。

```
test=# \h CREATE DOMAIN
Command: CREATE DOMAIN
Description: define a new domain
Syntax:
CREATE DOMAIN name [ AS ] data_type
[ COLLATE collation ]
[ DEFAULT expression ]
[ constraint [ ... ] ]
where constraint is:
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
URL: https://www.postgresql.org/docs/13/sql-createdomain.html
```

在这个例子中，PL/Perl函数将被用来创建一个叫做email的域，而这个域又可以作为一个数据类型使用。下面的代码显示了如何创建该域。

```
test=# CREATE DOMAIN email AS text
  CHECK (verify_email(VALUE) = true);
CREATE DOMAIN
```

CREATE DOMAIN命令创建了额外的类型，并自动应用检查约束，以确保该限制在整个数据库中被一致使用。

正如我们之前提到的，域的功能就像一个正常的数据类型。

```
test=# CREATE TABLE t_email (id serial, data email);
CREATE TABLE
```

这个Perl函数确保没有任何违反我们检查的东西可以被插入数据库，下面的例子成功地证明了这一点。

```
test=# INSERT INTO t_email (data)
VALUES ('somewhere@example.com');
INSERT 0 1
test=# INSERT INTO t_email (data)
VALUES ('somewhere_wrong_example.com');
ERROR: value for domain email violates check constraint "email_check"
```

Perl可能是一个很好的选择，可以进行字符串计算，但是，像往常一样，你必须决定你是否想让这些代码直接进入数据库，或者不直接进入。

2.2.2 在 PL/Perl 和 PL/PerIu 之间做出选择

到目前为止，Perl代码还没有引起任何与安全有关的问题，因为我们所做的只是使用正则表达式。这里的问题是，如果有人试图在Perl函数里面做一些讨厌的事情怎么办？正如我们已经说过的，PL/Perl会简单地出错，你可以在下一个列表中看到。

```
test=# CREATE OR REPLACE FUNCTION test_security()
RETURNS boolean AS
$$
use strict;
my $fp = open("/etc/password", "r");
return false;
$$ LANGUAGE 'plperl';
ERROR: 'open' trapped by operation mask at line
CONTEXT: compilation of PL/Perl function "test_security"
```

该列表显示，PL/Perl在你试图创建函数时就会报错。一个错误会立即显示出来。如果你真的想在Perl中运行不受信任的代码，你必须使用PL/PerIu。

```
test=# CREATE OR REPLACE FUNCTION first_line()
RETURNS text AS
$$
open(my $fh, '<:encoding(UTF-8)', "/etc/passwd")
or elog(NOTICE, "Could not open file '$filename' $!");
my $row = <$fh>;
close($fh);
return $row;
$$ LANGUAGE 'plperlu';
CREATE FUNCTION
```

该程序保持不变。它返回一个字符串。然而，它被允许做任何事情。唯一的区别是，该函数被标记为plperlu。

结果有些出乎意料

```
test=# SELECT first_line();
first_line
-----
root:x:0:0:root:/root:/bin/bash+
(1 row)
```

第一行将按预期显示。

2.2.3 使用 SPI 接口

偶尔，你的Perl程序要进行一些数据库工作。记住，这个函数是数据库连接的一部分。因此，实际创建一个数据库连接是没有意义的。为了与数据库对话，PostgreSQL服务器基础设施提供了服务器编程接口(SPI)，这是一个C接口，你可以用它来与数据库内部对话。所有帮助你运行服务器端代码的过程性语言都使用这个接口来向你暴露功能。PL/Perl也是这样做的，在本节中，你将学习如何使用SPI接口周围的Perl包装器。

你可能想做的最重要的事情是简单地运行SQL，并检索出被取走的行数。spi_exec_query函数正是用来做这个的。传递给该函数的第一个参数是查询参数。第二个参数是你实际想要检索的行数。为了简单起见，我决定获取所有的行。下面的代码显示了一个例子。

```
test=# CREATE OR REPLACE FUNCTION spi_sample(int)
RETURNS void AS
$$
my $rv = spi_exec_query(" SELECT *
  FROM generate_series(1, $_[0])", $_[0]
);
elog(NOTICE, "rows fetched: " . $rv->{processed});
elog(NOTICE, "status: " . $rv->{status});
return;
$$ LANGUAGE 'plperl';
```

SPI将执行查询并显示行数。这里重要的是，所有的存储过程语言都提供了一个发送日志信息的方法。在PL/Perl的情况下，这个函数被称为elog，它需要两个参数。第一个参数定义了消息的重要性(INFO, NOTICE, WARNING, ERROR, 等等)，第二个参数包含了实际的消息。

下面的消息显示了查询返回的内容。

```
test=# SELECT spi_sample(9);
NOTICE: rows fetched: 9
NOTICE: status: SPI_OK_SELECT
spi_sample
-----
(1 row)
```

对SPI的调用工作得很好，并且显示了状态。

2.2.4 使用 SPI 设置返回函数

在许多情况下，你并不只是想执行一些SQL语句，然后忘记它。在大多数情况下，一个存储过程会在结果上循环，并对其进行处理。下面的例子将展示你如何在一个查询的输出上进行循环。除此之外，我还决定加强这个例子，让这个函数返回一个复合数据类型。在Perl中使用复合类型是非常容易的，因为你可以简单地将数据塞进一个哈希值并返回。

return_next函数将逐渐建立起结果集，直到函数以一个简单的返回语句结束。

下面代码中的例子生成了一个由随机值组成的表。

```
CREATE TYPE random_type AS (a float8, b float8);
CREATE OR REPLACE FUNCTION spi_srf_perl(int)
RETURNS setof random_type AS
$$
my $rv = spi_query("SELECT random() AS a,
random() AS b
FROM generate_series(1, $_[0])");
while (defined (my $row = spi_fetchrow($rv)))
{
    elog(NOTICE, "data: " .
$row->{a} . " / " . $row->{b});
    return_next({a => $row->{a},
b => $row->{b}});
}
return;
$$ LANGUAGE 'plperl';
CREATE FUNCTION
```

首先，spi_query函数被执行，然后使用spi_fetchrow函数启动一个循环。在这个循环中，复合类型将被组装起来并塞进结果集中。正如预期的那样，该函数将返回一组随机值。

```
test=# SELECT * FROM spi_srf_perl(3);
NOTICE: data: 0.154673356097192 / 0.278830723837018
CONTEXT: PL/Perl function "spi_srf_perl"
NOTICE: data: 0.615888888947666 / 0.632620786316693
CONTEXT: PL/Perl function "spi_srf_perl"
NOTICE: data: 0.910436692181975 / 0.753427186980844
CONTEXT: PL/Perl function "spi_srf_perl"
a_col | b_col
-----+
0.154673356097192 | 0.278830723837018
0.615888888947666 | 0.632620786316693
0.910436692181975 | 0.753427186980844
(3 rows)
```

请记住，返回集合的函数必须是物化的，以便整个结果集可以被存储在内存中。

2.2.5 在 PL/Perl 中转义和支持函数

到目前为止，我们只使用了整数，所以SQL注入或特殊表名都不是问题。基本上，可以使用以下函数。

- quote_literal: quote_nullable: 返回一个字符串的引用，作为一个字符串字面。引用一个字符串。
- quote_ident: 引述SQL标识符（对象名称等）。
- decode_bytea: 对PostgreSQL的字节数组字段进行解码。
- encode_bytea: 对数据进行编码，并把它变成一个字节数组。
- encode_literal_array: 这是对一个字词阵列的编码。
- encode_typed_literal: 将一个Perl变量转换为作为第二个参数传递的数据类型的值，并返回该值的字符串表示。
- encode_array_constructor: 将引用的数组的内容以数组构造器的格式返回为字符串。
- looks_like_number: 如果一个字符串看起来像一个数字，则返回true。
- is_array_ref: 如果某个东西是一个数组引用，则返回true。

这些函数总是可用的，它们可以被直接调用，而不需要包含一个库。

2.2.6 跨函数调用共享数据

有时，有必要在不同的通话中分享数据。基础设施有办法真正做到这一点。在Perl中，一个哈希值可以用来存储任何需要的数据。看一下下面的例子吧。

```
CREATE FUNCTION perl_shared(text) RETURNS int AS
$$
if ( !defined ${_SHARED{$_[0]}} )
{
    ${_SHARED{$_[0]}} = 0;
}
else
{
    ${_SHARED{$_[0]}}++;
}
return ${_SHARED{$_[0]}};
$$ LANGUAGE 'plperl';
```

一旦我们发现传给函数的键还不在那里，\$_SHARED变量就会被初始化为0。对于其他的调用，1会被添加到计数器中，留给我们的是以下输出。

```
test=# SELECT perl_shared('some_key') FROM generate_series(1, 3);
perl_shared
-----
0
1
2
(3 rows)
```

如果是比较复杂的语句，开发者通常不知道函数将以什么顺序被调用。牢记这一点很重要。在大多数情况下，你不能依赖执行顺序。

2.2.7 在 Perl 中编写触发器

每一种与PostgreSQL核心一起提供的存储过程语言都允许你用该语言编写触发器。当然，这也适用于Perl。由于本章篇幅有限，我决定不包括用Perl编写触发器的例子，而是让你看PostgreSQL的官方文档：<https://www.postgresql.org/docs/10/static/plperl-triggers.html>。基本上，用Perl写一个触发器和用PL/pgSQL写一个触发器没有什么区别。所有预定义的变量都到位了，至于返回值，规则适用于每一种存储过程语言。

2.3 介绍 PL/Python

如果你碰巧不是Perl专家，PL/Python可能是适合你的东西。Python已经成为PostgreSQL基础设施的一部分很长时间了，因此是一个坚实的、经过良好测试的实现。

谈到PL/Python，有一件事你必须牢记。PL/Python只能作为一种不受信任的语言使用。从安全的角度来看，在任何时候都必须记住这一点。

要启用PL/Python，你可以从你的命令行中运行以下一行，并测试你想用PL/Python使用的数据库的名称：

```
create lang plpythonu test
```

一旦语言被启用，就可以写代码了。

另外，你也可以使用CREATE LANGUAGE子句。另外，请记住，为了使用服务器端语言，需要包含这些语言的PostgreSQL包（postgresql-plpython-\$(VERSIONNUMBER)，等等）。

2.3.1 编写简单的 PL/Python 代码

在本节中，你将学习如何编写简单的Python程序。我们在这里要讨论的例子很简单：如果你在奥地利开车拜访客户，你可以在每公里的费用中扣除42欧分，以减少你的所得税。因此，该函数所做的事是，获取公里数，并返回我们可以从税单中扣除的金额。下面是它的工作原理。

```
CREATE OR REPLACE FUNCTION calculate_deduction(km float)
RETURNS numeric AS
$$
if km <= 0:
    elog(ERROR, 'invalid number of kilometers')
else:
    return km * 0.42
$$ LANGUAGE 'plpythonu';
```

该函数确保只接受正值。最后，结果被计算并返回。正如你所看到的，Python函数传递给PostgreSQL的方式与Perl或PL/pgSQL并没有什么不同。

2.3.2 使用 SPI 接口

像所有程序性语言一样，PL/Python让你可以访问SPI接口。下面的例子显示了如何将数字加起来。

```
CREATE FUNCTION add_numbers(rows_desired integer)
RETURNS integer AS
$$
mysum = 0
```

```

cursor = plpy.cursor("SELECT * FROM
generate_series(1, %d) AS id" % (rows_desired))
while True:
    rows = cursor.fetch(rows_desired)
    if not rows:
        break
    for row in rows:
        mysum += row['id']
return mysum
$$ LANGUAGE 'plpythonu';

```

当你尝试这个例子时，要确保对光标的调用实际上是单行的。Python 是关于缩进的，所以如果你的代码是由一行或两行组成的，确实有区别。

一旦游标被创建，我们就可以在它上面循环，把这些数字加起来。这些行里面的列可以很容易地用列名来引用。

调用该函数将返回所需的结果。

```

test=# SELECT add_numbers(10);
add_numbers
-----
55
(1 row)

```

如果你想检查一个SQL语句的结果集，PL/Python提供了各种函数，允许你从结果中检索更多的信息。同样，这些函数是SPI在C语言层面上提供的包装。下面的函数更仔细地检查了一个结果。

```

CREATE OR REPLACE FUNCTION result_diag(rows_desired integer)
RETURNS integer AS
$$
rv = plpy.execute("SELECT *
FROM generate_series(1, %d) AS id" % (rows_desired))
plpy.notice(rv.nrows())
plpy.notice(rv.status())
plpy.notice(rv.colnames())
plpy.notice(rv.coltypes())
plpy.notice(rv.coltypmods())
plpy.notice(rv.str())
return 0
$$ LANGUAGE 'plpythonu';

```

`nrows()`函数将显示行的数量。`status()`函数告诉我们一切工作是否顺利。`colnames()`函数返回一个列的列表。`coltypes()`函数返回结果集中数据类型的对象ID。23是整数的内部数字，如下面的代码所示。

```

test=# SELECT typname FROM pg_type WHERE oid = 23;
typname
-----
int4
(1 row)

```

然后是`tymod`。考虑像`varchar(20)`这样的东西：类型的配置部分就是`tymod`的全部内容。最后，有一个函数将整个东西作为一个字符串返回，用于调试目的。调用该函数将返回以下结果。

```
test=# SELECT result_diag(3);
NOTICE: 3
NOTICE: 5
NOTICE: ['id']
NOTICE: [23]
NOTICE: [-1]
NOTICE: <PLyResult status=5 nrows=3 rows=[{'id': 1,
{'id': 2}, {'id': 3}]>
result_diag
-----
0
(1 row)
```

该清单显示了我们的诊断函数返回的内容。在SPI接口中还有很多函数可以帮助你执行SQL。

2.3.3 处理错误

偶尔，你可能要捕捉一个错误。当然，这在Python中也是可能的。下面的例子显示了这是如何进行的。

```
CREATE OR REPLACE FUNCTION trial_error()
RETURNS text AS
$$
try:
    rv = plpy.execute("SELECT surely_a_syntax_error")
except plpy.SPIError:
    return "we caught the error" else:
else:
    return "all fine"
$$ LANGUAGE 'plpythonu';
```

你可以使用一个正常的try或except块，并访问plpy来处理你想捕捉的错误。然后该函数可以正常返回而不破坏你的事务，如下所示。

```
test=# SELECT trial_error();
trial_error
-----
we caught the error
(1 row)
```

记住，PL/Python可以完全访问PostgreSQL的内部结构。因此，它也可以将各种错误暴露给你的程序。下面是一个例子。

```
except spiexceptions.DivisionByZero:
    return "found a division by zero"
except spiexceptions.UniqueViolation:
    return "found a unique violation"
except plpy.SPIError, e:
    return "other error, SQLSTATE %s" % e.sqlstate
```

这段代码显示了如何捕捉各种Python错误。在Python中捕捉错误真的很容易，而且可以帮助防止你的函数失败。在这一节中，你已经了解了Python的错误处理。在下一节中，我们将深入研究，看看我们如何帮助优化器。

3 改进函数

到目前为止，你已经看到如何用各种语言编写基本函数和触发器。当然，还有很多语言被支持。其中最突出的是PL/R（R是一个强大的统计包）和PL/v8（它是基于谷歌JavaScript引擎的）。然而，这些语言已经超出了本章的范围（不管它们是否有用）。

在本节中，我们将重点讨论如何提高一个函数的性能。有几种方法可以让我们加快处理速度。

- 减少函数调用次数
- 使用缓存计划
- 给优化器以提示

在本节中，将讨论所有这三个方面。让我们从减少函数调用的数量开始。

3.1 减少函数调用次数

在很多情况下，性能不好是因为函数被调用得太频繁了。在我看来--我怎么强调这一点都不为过--太频繁地调用东西是性能不好的主要原因。当你创建一个函数时，你可以从三种类型的函数中选择：易变的、稳定的和不可变的。下面是一个例子。

```
test=# SELECT random(), random();
random | random
+-----+
0.276252629235387 | 0.710661871358752
(1 row)

test=# SELECT now(), now();
now | now
+-----+
2020-09-01 09:58:33.153834+02 | 2020-09-01 09:58:33.153834+02
(1 row)

test=# SELECT pi();
pi
+-----+
3.14159265358979
(1 row)
```

一个易失性函数意味着该函数不能被优化掉。它必须一次又一次地被执行。一个易失性函数也可能是某个索引不被使用的原因。默认情况下，每个函数都被认为是不稳定的。一个稳定的函数将总是在同一个事务中返回相同的数据。它可以被优化，调用也可以被删除。now()函数就是一个稳定函数的好例子；在同一个事务中，它返回相同的数据。不可变的函数是黄金标准，因为它们允许大多数优化，这是因为如果给它们相同的输入，它们总是返回相同的结果。作为优化函数的第一步，一定要确保它们被正确标记，在定义的末尾加上volatile、stable或immutable。

在下一小节中，你将了解到缓存计划。

3.2 使用缓存计划

在PostgreSQL中，一个查询是通过四个阶段来执行的。

1. 解析器。它检查语法。
2. 重写系统。它负责处理规则。
3. 优化器/规划器。这将优化查询。
4. 执行器。执行由计划器提供的计划。

如果查询时间较短，前三个步骤相对于实际执行时间来说是比较耗时的。因此，对执行计划进行缓存是有意义的。PL/pgSQL基本上在幕后为你自动完成所有的计划缓存。你不需要担心这个问题。PL/Perl和PL/Python将为你提供选择。

SPI接口提供了一些函数，这样你就可以处理和运行准备好的查询，所以程序员可以选择是否应该准备好一个查询。在长查询的情况下，使用未准备的查询实际上是有意义的。短的查询应该总是准备好的，以减少内部开销。

3.3 将成本分配给函数

从优化器的角度来看，一个函数基本上就像一个运算符。PostgreSQL也会以相同的方式处理成本，就像它是一个标准运算符一样。问题是这样的：两个数字相加通常比使用PostGIS提供的函数相交成本线要便宜。问题是，优化器不知道一个函数是便宜还是昂贵。

幸运的是，我们可以告诉优化器使函数更便宜或更昂贵。下面是CREATE FUNCTION的语法。

```
test=# \h CREATE FUNCTION
Command: CREATE FUNCTION
Description: Define a new function
Syntax:
CREATE [ OR REPLACE ] FUNCTION
...
| COST execution_cost
| ROWS result_rows
...
```

COST参数表示你的操作符比标准操作符真正昂贵的程度。它是cpu_operator_cost的一个乘数，不是一个静态值。一般来说，默认值是100，除非该函数是用C语言编写的。现在我们已经了解了所有关于函数的知识，让我们在下一节中进一步探讨它们。

4 为各种目的使用函数

在PostgreSQL中，存储过程几乎可以用于一切。在本章中，你已经了解了CREATE DOMAIN子句等，但也可以创建你自己的操作符、类型转换，甚至排序规则。

在本节中，你将看到如何创建一个简单的类型转换，以及如何利用它来发挥你的优势。要定义一个类型转换，可以考虑看一下CREATE CAST子句。这个命令的语法在下面的代码中显示。

```
test=# \h CREATE CAST
Command: CREATE CAST
Description: define a new cast
Syntax:
CREATE CAST (source_type AS target_type)
    WITH FUNCTION function_name [ (argument_type [, ...]) ]
    [ AS ASSIGNMENT | AS IMPLICIT ]
CREATE CAST (source_type AS target_type)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]
CREATE CAST (source_type AS target_type)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

URL: <https://www.postgresql.org/docs/13/sql-createcast.html>

使用这个东西是非常简单的。你只需告诉PostgreSQL它应该调用哪个程序，以便将任何类型的数据转换为你想要的数据类型。

在标准的PostgreSQL中，你不能把IP地址转为布尔值。因此，它是一个很好的例子。首先，存储过程必须被定义。

```
CREATE FUNCTION inet_to_boolean/inet)
RETURNS boolean AS
$$
BEGIN
RETURN true;
END;
$$ LANGUAGE 'plpgsql';
```

为了简单起见，它返回真。然而，你可以使用任何语言的任何代码来进行实际转换。

在下一步，已经可以定义CAST类型了。

```
CREATE CAST (inet AS boolean)
WITH FUNCTION inet_to_boolean/inet) AS IMPLICIT;
```

我们需要做的第一件事是告诉PostgreSQL，我们要把inet转换为boolean。然后，函数被列出，我们告诉PostgreSQL，我们更喜欢隐式转换。这是一个简单明了的过程，我们可以按照下面的方法来测试转换。

```
test=# SELECT '192.168.0.34'::inet::boolean;
bool
-----
t
(1 row)
```

类型案例是成功的。基本上，同样的逻辑也可以应用于定义整理。同样，可以用一个存储过程来执行任何需要做的事情。

```
test=# \h CREATE COLLATION
Command: CREATE COLLATION
Description: define a new collation
Syntax:
CREATE COLLATION [ IF NOT EXISTS ] name (
[ LOCALE = locale, ]
[ LC_COLLATE = lc_collate, ]
[ LC_CTYPE = lc_ctype, ]
[ PROVIDER = provider, ]
[ DETERMINISTIC = boolean, ]
[ VERSION = version ]
)
CREATE COLLATION [ IF NOT EXISTS ] name FROM existing_collation
URL: https://www.postgresql.org/docs/13/sql-createcollation.html
```

CREATE COLLATION的语法真的很简单。虽然创建排序是可能的，但它仍然是很少使用的功能之一。

存储过程和函数提供了很多东西。许多事情都是可能的，而且可以用一种漂亮而有效的方式来完成。

5 总结

在本章中，你已经学会了如何编写存储过程。在理论介绍之后，我们的注意力集中在PL/pgSQL的一些精选特性上。除此之外，你还学习了如何使用PL/Perl和PL/Python，这是PostgreSQL提供的两种重要语言。当然，还有更多的语言可以使用。但是，由于本书范围的限制，无法详细介绍它们。如果你想了解更多，请查看以下网站：https://wiki.postgresql.org/wiki/PL_Matrix。我们还学习了如何改进函数调用，以及如何将其用于其他各种用途，以加快应用程序的速度，并做更多的事情。

在第8章，管理PostgreSQL的安全，你将学习PostgreSQL的安全。你将学习如何在总体上管理用户和权限。在此基础上，你还将学习网络安全。

6 问题

- 函数和存储过程之间的区别是什么？
- 受信任的语言和不受信任的语言之间的区别是什么？
- 一般来说，函数是好是坏？
- 在PostgreSQL中哪些服务器端语言是可用的？
- 什么是触发器？
- 哪些语言可以用来编写函数？
- 哪种语言是最快的？

这些问题的答案可以在 GitHub 仓库中找到 (<https://github.com/PacktPublishing/Mastering-PostgreSQL-13-Fourth-Edition>)

第八章 管理PG安全

第八章 管理PG安全

- 1.管理网络安全
- 2.了解绑定地址和连接
- 3.检查连接和性能
- 4.生活在没有 TCP 的世界里
- 5.管理pg_hba.conf
- 6.处理ssl
- 7.处理实例级安全
- 8.创建和修改用户
- 9.定义数据库级安全性
- 10.调整模式层面的权限
- 11.使用表格
- 12.处理列级安全
- 13.配置默认权限
- 14.深入研究 RLS
- 15.检查权限
- 16.重新分配对象和删除用户
- 17.总结
- 18.问题

在第7章，编写存储过程中，我们学习了存储过程和编写服务器端代码。在介绍了许多重要的主题之后，现在是时候转向PostgreSQL的安全问题了。在这里，我们将学习如何保证服务器的安全，并配置权限以避免安全宣扬。

本章将涉及以下主题。

管理网络安全

挖掘行级安全 (RLS)

检查权限

重新分配对象和放弃用户

在本章结束时，我们将能够专业地配置PostgreSQL安全。现在让我们从管理网络安全开始吧。

1.管理网络安全

在继续讨论现实世界的实际例子之前，让我们简单地关注一下我们将要处理的各种安全层。在处理安全问题时，为了有组织地处理与安全有关的问题，牢记这些层次是有意义的。

以下是我的思想模型：

- 绑定地址：postgresql.conf文件中的listen_addresses
- 基于主机的访问控制：pg_hba.conf文件
- 实体级权限：用户、角色、数据库创建、登录和复制
- 数据库级权限：连接、创建模式等等
- 模式级权限：使用模式和在模式内创建对象

- 表级权限：选择、插入、更新，以及更多
- 列级权限：允许或限制对列的访问
- RLS：限制对行的访问

为了读取一个值，PostgreSQL 必须确保我们在每个级别上都有足够的权限。整个权限链必须正确。多年来，我的小模型一直很好地帮助我在实际应用程序中反复调试与安全相关的问题。它有望帮助您使用更系统的方法，从而提高安全性。

2.了解绑定地址和连接

在配置PostgreSQL服务器时，首先需要做的一件事是定义远程访问。默认情况下，PostgreSQL不接受远程连接。这里重要的是，PostgreSQL甚至不拒绝连接，因为它根本不监听端口。如果我们尝试连接，错误信息实际上将来自操作系统，因为PostgreSQL根本不关心。

假设在192.168.0.123上有一个使用默认配置的数据库服务器，将发生以下情况。

```
iMac:~ hs$ telnet 192.168.0.123 5432
Trying 192.168.0.123...
telnet: connect to address 192.168.0.123: Connection refused
telnet: Unable to connect to remote host
```

Telnet试图在5432端口建立一个连接，但立即被远程拒绝。从外面看，好像PostgreSQL根本就没有运行。

在postgresql.conf文件中可以找到成功的关键。

```
# - Connection Settings -
# listen_addresses = 'localhost'
# what IP address(es) to listen on;
# comma-separated list of addresses;
# defaults to 'localhost'; use '*' for all
# (change requires restart)
```

listen_addresses设置将告诉PostgreSQL要听哪些地址。从技术上讲，这些地址是绑定地址。这实际上是什么意思呢？假设我们的机器上有四个网卡。我们可以监听，比如，其中三个互联网协议（IP）地址。PostgreSQL会考虑到对这三块网卡的请求，而不会去监听第四块网卡。该端口被简单地关闭。

我们必须把我们服务器的IP地址放到listen_addresses中，而不是客户的IP。

如果我们在PostgreSQL中加入*，我们将监听分配给你的机器的每个IP。

请记住，改变listen_addresses需要重新启动PostgreSQL服务。如果不重启，就不能临时改变它。

然而，还有更多与连接管理有关的设置是需要了解，并且非常重要。它们如下：

```
#port = 5432
# (change requires restart)
max_connections = 100
# (change requires restart)
# Note: Increasing max_connections costs ~400 bytes of
# shared memory per
# connection slot, plus lock space
# (see max_locks_per_transaction)
#superuser_reserved_connections = 3
# (change requires restart)
#unix_socket_directories = '/tmp'
```

```
# comma-separated list of directories
# (change requires restart)
#unix_socket_group =
# (change requires restart)
#unix_socket_permissions = 0777
# begin with 0 to use octal notation
# (change requires restart)
```

首先，PostgreSQL只监听一个传输控制协议（TCP）端口，其默认值是5432。请记住，PostgreSQL将只监听一个端口。每当有请求进入时，Postmaster将分叉并创建一个新的进程来处理该连接。默认情况下，最多允许100个正常连接。在此基础上，为超级用户保留3个额外的连接。这意味着，我们可以有97个连接，加上3个超级用户，或者100个超级用户连接。为了开始工作，我们将首先看一下连接和性能

请注意，这些与连接有关的设置也需要重新启动。其原因是，共享内存分配了一个静态的内存量，不能临时改变。

3.检查连接和性能

在做咨询的时候，很多人问我提高连接限制是否会对一般的性能产生影响。答案是不多，因为由于上下文切换，总是有一些开销。至于有多少个连接，这没有什么区别。然而，有区别的开放快照的数量。开放的快照越多，数据库方面的开销就越大。换句话说，我们可以廉价地增加max_connections。在下一节，我们将学习如何出于安全原因避免使用TCP。

如果你对一些真实世界的数据感兴趣，可以考虑看一下 https://www.cybertec-postgresql.com/en/max_connections-performance-impacts/

4.生活在没有 TCP 的世界里

在某些情况下，我们可能不希望使用网络。经常发生的情况是，无论如何，数据库只能与本地应用程序对话。也许我们的PostgreSQL数据库已经和我们的应用程序一起被运送了，或者我们只是不想承担使用网络的风险。在这种情况下，Unix套接字是你需要的。Unix套接字是一种无网络的通信方式。你的应用程序可以通过Unix套接字在本地进行连接，而不向外界暴露任何信息。

然而，我们需要的是一个目录。默认情况下，PostgreSQL 将使用 /tmp 目录。但是，如果每台机器运行着多个数据库服务器，则每个服务器都需要一个单独的数据目录来存放

除了安全之外，不使用网络可能是个好主意的原因还有很多。其中一个原因是性能。使用 Unix 套接字比通过环回设备 (127.0.0.1) 快很多。如果这听起来令人惊讶，请不要担心；它适用于很多人。但是，如果您只运行非常小的查询，则不应低估实际网络连接的开销。

为了说明这到底意味着什么，我包括了一个简单的基准。

我们将创建一个script.sql文件。这是一个简单的脚本，用于创建一个随机数并进行选择。这是最简单的语句。没有什么比获取一个数字更简单的了。

因此，让我们在一台普通的笔记本电脑上运行这个简单的基准测试。为此，我们将写一个叫做script.sql的小东西。它将被下面的基准测试所使用。

```
[hs@linuxpc ~]$ cat /tmp/script.sql
SELECT 1
```

然后，我们可以简单地运行 pgbench 来一遍又一遍地执行 SQL。-f 选项允许我们将 SQL 的名称传递给脚本。-c 10 表示我们希望处于活动状态 5 秒 (-T 5) 内有 10 个并发连接。基准测试以 postgres 用户身份运行，并且应该使用默认情况下应该存在的 postgres 数据库。请注意，以下示例适用于 Red Hat Enterprise Linux (RHEL) 衍生产品。基于 Debian 的系统将使用不同的路径：

```
[hs@linuxpc ~]$ pgbench -f /tmp/script.sql -c 10 -T 5 -U postgres postgres 2> /dev/null
transaction type: /tmp/script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
duration: 5 s
number of transactions actually processed: 871407
latency average = 0.057 ms
tps = 174278.158426 (including connections establishing)
tps = 174377.935625 (excluding connections establishing)
```

我们可以看到，没有向pgbench传递主机名，因此该工具在本地连接到Unix套接字，并尽可能快地运行脚本。在这个四核英特尔盒子上，系统能够实现每秒约174,000次交易

如果加上-h localhost会怎样？性能将发生变化，正如你在下面的代码段中看到的那样。

```
[hs@linuxpc ~]$ pgbench -f /tmp/script.sql -h localhost -c 10 -T 5 -U postgres
postgres 2> /dev/null
transaction type: /tmp/script.sql
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
duration: 5 s
number of transactions actually processed: 535251
latency average = 0.093 ms
tps = 107000.872598 (including connections establishing)
tps = 107046.943632 (excluding connections establishing)
```

吞吐量将像石头一样下降到每秒107000个交易。这种差异显然与网络开销有关。

通过使用-j选项（分配给pgbench的线程数），我们可以从我们的系统中挤出一些更多的事务。然而，在我们的情况下，这并没有改变基准测试的整体情况。在其他测试中，它确实如此，因为如果你不提供足够的CPU能力，pgbench可能是一个真正的瓶颈

正如我们所看到的，网络不仅可以是一个安全问题，也是一个性能问题。然而，性能并不是唯一重要的方面。由于我们在这里主要讨论的是安全问题，因此下一个难题是关于 pg_hba.conf 的。

5.管理pg_hba.conf

在配置完绑定地址后，我们可以进入下一个层次。pg_hba.conf文件将告诉PostgreSQL如何对通过网络来的人进行认证。一般来说，pg_hba.conf文件的条目有如下布局。

```
# Local DATABASE USER METHOD [OPTIONS]
# host DATABASE USER ADDRESS METHOD [OPTIONS]
# hostssl DATABASE USER ADDRESS METHOD [OPTIONS]
# hostnoss1 DATABASE USER ADDRESS METHOD [OPTIONS]
# hostgssenc DATABASE USER ADDRESS METHOD [OPTIONS]
# hostnogssenc DATABASE USER ADDRESS METHOD [OPTIONS]
```

有四种类型的规则可以放入pg_hba.conf文件中：

- local:这可以用来配置本地Unix套接字连接。
- host:这可以用于安全套接字层（SSL）和非SSL连接。

- hostssl: 这只对SSL连接有效。要利用这个选项，SSL必须被编译到服务器中，如果我们使用PostgreSQL的预包装版本，就会出现这种情况。除此之外，ssl = on必须在postgresql.conf文件中设置。这个文件在服务器启动时被调用。
- hostnossal: 这适用于非SSL连接。
- hostgssenc: 这个规则定义了只有在可以进行GSSAPI加密的情况下才会创建一个连接。否则就会失败。
- hostnogssenc: 这条规则与hostgssenc完全相反。

可以将规则列表合并到 pg_hba.conf 文件中。这是一个例子：

```
# TYPE DATABASE USER ADDRESS METHOD
# "local" is for Unix domain socket connections only
local all all trust
# IPv4 local connections:
host all all 127.0.0.1/32 trust
# IPv6 local connections:
host all all ::1/128 trust
```

您可以看到三个简单的规则：

- 本地记录说，来自本地Unix套接字的所有数据库的用户都要被信任。信任方法意味着不需要向服务器发送密码，人们可以直接登录
- 另外两条规则说，同样适用于来自127.0.0.1 localhost和::1/128的连接，后者是一个IPv6地址

由于没有密码的连接肯定不是远程访问的最佳选择，PostgreSQL提供了各种认证方法，可以用来灵活地配置pg_hba.conf文件。下面是一个可能的认证方法的列表：

- trust: 这允许认证而不提供密码。所需的用户必须在PostgreSQL端可用。
- reject: 连接将被拒绝。
- md5和password: 连接可以使用密码来创建。md5意味着密码在通过网络发送时被加密了。在密码的情况下，凭证是以明文形式发送的，在现代系统中不应再这样做，md5已不再被认为是安全的。你应该使用scram-sha-256来代替PostgreSQL 10及以后的版本。
- scram-sha-256: 这个设置是md5的继承者，使用的哈希值比以前的版本要安全得多。
- gss and sspi: 这使用通用安全服务应用程序接口（GSSAPI）或安全支持提供者接口（SSPI）认证。这可能用于TCP/IP连接。这里的想法是允许单点登录。
- ident: 这通过联系客户的身份服务器获得客户的操作系统用户名，并检查它是否与请求的数据库用户名相匹配。
- Peer: 假设我们在Unix上以abc的身份登录。如果peer被启用，我们只能以abc的身份登录PostgreSQL。如果我们试图改变用户名，我们将被拒绝。美中不足的是，abc不需要密码就可以进行验证。这里的想法是，只有数据库管理员可以在Unix系统上登录数据库，而不是其他只是拥有密码的人或同一台机器上的Unix帐户。这只对本地连接起作用，
- pam。这使用了可插拔认证模块（PAM）。如果你想使用PostgreSQL不提供的认证方式，这一点特别重要。要使用PAM，在你的Linux系统上创建一个叫做/etc/pam.d/postgresql的文件，并把你打算使用的PAM模块放到配置文件中。使用PAM，我们甚至可以针对不太常见的组件进行认证。然而，它也可以用来连接到活动目录等。
- ldap: 这个配置允许你使用轻量级目录访问协议（LDAP）进行认证。注意，PostgreSQL只要求LDAP进行认证；如果一个用户只存在于LDAP端，而不存在于PostgreSQL端，你就不能登录。你还应该注意，PostgreSQL必须知道你的LDAP服务器在哪里。所有这些信息都必须存储在pg_hba.conf文件中，如官方文档<https://www.postgresql.org/docs/10/static/auth-methods.html#AUTH-LDAP>。
- radius: 远程认证拨入用户服务（RADIUS）是一种执行单点登录的方法。同样，参数是使用配置选项传递的。
- cert: 这种认证方法使用SSL客户证书来执行认证，因此只有在使用SSL的情况下才有可能。这里的优点是不需要发送密码。证书的CN属性将与请求的数据库用户名进行比较，如果它们匹配，将允

许登录。可以使用一个地图来允许用户的映射

规则可以简单地一个接一个地列出。这里重要的是，顺序确实有区别，如下面的例子所示。

```
host all all 192.168.1.0/24 scram-sha-256
host all all 192.168.1.54/32 reject
```

当PostgreSQL浏览pg_hba.conf文件时，它将使用第一个匹配的规则。因此，如果我们的请求来自192.168.1.54，在我们进入第二条规则之前，第一条规则总是会匹配。这意味着，如果密码和用户都是正确的，192.168.1.54将能够登录；因此，第二条规则是没有意义的。

如果我们想排除IP，我们需要确保这两条规则被调换。在下一节中，我们将看一下SSL，以及你如何能够轻松地使用它

6.处理ssl

PostgreSQL允许在服务器和客户端之间的传输是加密的。加密是非常有益的，特别是在我们进行长距离通信的时候。SSL提供了一个简单而安全的方法，确保没有人能够监听你的通信。

在本节中，我们将学习如何设置 SSL：

- 首先要做的是在服务器启动时，在postgresql.conf文件中将ssl参数设置为on。在下一步，我们可以把SSL证书放到\$PGDATA目录中。如果我们不希望证书放在其他目录中，我们需要改变以下参数

```
#ssl_cert_file = 'server.crt' # (change requires restart)
#ssl_key_file = 'server.key' # (change requires restart)
#ssl_ca_file = '' # (change requires restart)
#ssl_crl_file = '' # (change requires restart)
```

- 如果我们想使用自签名的证书，我们需要执行以下步骤

```
openssl req -new -text -out server.req
```

回答OpenSSL所问的问题。确保我们输入本地主机名作为公共名称。我们可以把密码留空。这个调用将生成一个受密码保护的密钥；它不接受长度少于四个字符的密码。

- 要删除口令（如果你想自动启动服务器，你必须这样做），请运行以下代码。

```
openssl rsa -in privkey.pem -out server.key
rm privkey.pem
```

- 输入旧的口令来解锁现有的密钥。现在，使用下面的代码把证书变成一个自签名的证书，并把密钥和证书复制到服务器要找的地方

```
openssl req -x509 -in server.req -text -key server.key -out server.crt
```

- 这样做之后，确保这些文件有正确的权限。

```
chmod og-rwx server.key
```

- 一旦在pg_hba.conf文件中放入适当的规则，我们就可以使用SSL连接到你的服务器。为了验证我们是否在使用SSL，可以考虑查看pg_stat_ssl函数。它将告诉我们每一个连接以及它是否使用了SSL，

并将提供一些关于加密的重要信息

```
test=# \d pg_stat_ssl
View "pg_catalog.pg_stat_ssl"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
pid | integer | | |
ssl | boolean | | |
version | text | | |
cipher | text | | |
bits | integer | | |
compression | boolean | | |
client_dn | text | | |
client_serial | numeric | | |
issuer_dn | text | | |
```

- 如果一个进程的ssl字段包含true, PostgreSQL会做我们所期望的事情

```
postgres=# select * from pg_stat_ssl;
-[ RECORD 1 ]
-----
pid | 20075
ssl | t
version | TLSv1.2
cipher | ECDHE-RSA-AES256-GCM-SHA384
bits | 256
compression | f
clientdn |
client_serial |
issuer_dn |
```

一旦你配置好了SSL, 就该看看实例级的安全问题了

7.处理实例级安全

到目前为止, 我们已经配置了绑定地址, 并且告诉PostgreSQL在哪些IP范围内使用哪种认证方式。到现在为止, 这些配置纯粹是与网络有关的。

在这一节, 我们可以把注意力转移到实例级的权限上。最重要的是要知道PostgreSQL中的用户是否存在于实例级别上。如果我们创建了一个用户, 它不仅仅是在一个数据库中可见, 它可以被所有的数据库看到。一个用户可能只有访问一个数据库的权限, 但用户基本上是在实例级创建的。

对于那些刚接触PostgreSQL的人来说, 还有一件事你应该记住: 用户和角色是同样的东西。CREATE ROLE和CREATE USER子句有不同的默认值(唯一的区别是, 角色默认不获得LOGIN属性), 但是, 归根结底, 用户和角色是一样的。因此, CREATE ROLE和CREATE USER子句支持非常相同的语法。下面的列表包含了CREATE USER的语法概述

```
postgres=# \h CREATE USER
Command: CREATE USER
Description: define a new database role
Syntax:
CREATE USER name [ [ WITH ] option [ ... ] ]
where option can be:
    SUPERUSER | NOSUPERUSER
```

```
| CREATEDB | NOCREATEDB
| CREATEROLE | NOCREATEROLE
| INHERIT | NOINHERIT
| LOGIN | NOLOGIN
| REPLICATION | NOREPLICATION
| BYPASSRLS | NOBYPASSRLS
| CONNECTION LIMIT connlimit
| [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
| VALID UNTIL 'timestamp'
| IN ROLE role_name [, ...]
| IN GROUP role_name [, ...]
| ROLE role_name [, ...]
| ADMIN role_name [, ...]
| USER role_name [, ...]
| SYSID uid
```

URL: <https://www.postgresql.org/docs/13/sql-createuser.html>

让我们逐一讨论这些语法元素。我们可以看到的第一件事是，一个用户可以是超级用户，也可以是普通用户。如果某人被标记为SUPERUSER，就不再有普通用户必须面对的任何限制。SUPERUSER可以随心所欲地删除对象（例如，数据库）。

下一个重要的事情是，创建一个新的数据库需要实例级别的权限。

注意，当有人创建一个数据库时，这个用户将自动成为数据库的所有者。

规则是这样的：创建者总是自动成为一个对象的所有者（除非另有指定，如可以用CREATE DATABASE子句来做）。这样做的好处是，对象所有者也可以再次放弃一个对象：

CREATEROLE或NOCREATEROLE子句定义了是否允许某人创建新用户/角色。

下一个重要的事情是INHERIT或NOINHERIT子句。如果设置了INHERIT子句（这是默认值），一个用户可以继承其他用户的权限。使用继承的权限允许我们使用角色，这是一种抽象权限的好方法。例如，我们可以创建bookkeeper这个角色，并使许多其他角色继承bookkeeper的权限。这个想法是，我们只需要告诉PostgreSQL一次，一个bookkeeper角色被允许做什么，即使我们有很多人从事会计工作。

LOGIN或NOLOGIN子句定义了一个角色是否被允许登录到实例。

请注意，LOGIN子句并不足以实际连接到数据库。要做到这一点，需要更多的权限a

在这一点上，我们正试图使其进入实例，这是通往实例内所有数据库的大门。让我们回到我们的例子：bookkeeper可能被标记为NOLOGIN，因为我们希望人们用他们的真实姓名登录。你所有的会计人员（比如说，Joe和Jane）可能被标记为LOGIN子句，但可以继承bookkeeper角色的所有权限。像这样的结构可以很容易地确保所有的bookkeeper将拥有相同的权限，同时确保他们的个人活动在各自的身份下被操作和记录

如果我们计划用流式复制来运行PostgreSQL，我们可以用超级用户来做所有的交易日志流。然而，从安全的角度来看，这并不推荐。为了保证我们不必成为超级用户来流式传输xlog，PostgreSQL允许我们给普通用户以复制的权利，然后可以用它来做流式传输。通常的做法是创建一个特殊的用户，专门用于管理流复制。

正如我们在本章后面将看到的，PostgreSQL提供了一个叫做行级安全（RLS）的功能。其原理是，我们可以将行从用户的范围内排除。如果一个用户明确应该绕过RLS，把这个值设置为BYPASSRLS。默认值是NOBYPASSRLS。

有时，限制一个用户允许的连接数是有意义的。CONNECTION LIMIT正是允许我们这样做的。注意，总的来说，连接数不可能超过postgresql.conf文件中定义的数量（max_connections）。然而，我们总是可以把某些用户限制在一个较低的数值上。

默认情况下，PostgreSQL会在系统表中存储加密的密码，这是一个很好的默认行为。然而，假设你正在做一个培训课程，有10个学生参加，每个人都连接到你的盒子上。你可以百分之百的肯定，这些人中有一个会偶尔忘记自己的密码。由于你的设置不是安全关键，你可能会决定以明文方式存储密码，这样你就可以很容易地查找它并把它交给学生。如果你正在测试软件，这个功能可能也会派上用场。

通常情况下，我们已经知道某人会很快离开组织。VALID UNTIL子句允许我们在一个特定用户的账户过期时自动锁定他们

IN ROLE子句列出了一个或多个现有的角色，新的角色将被立即添加为新的成员。这有助于避免额外的手工步骤。IN ROLE的一个替代方法是IN GROUP。

ROLE子句将定义被自动添加为新角色成员的角色。

ADMIN子句与ROLE子句相同，但它增加了WITH ADMIN OPTION。最后，我们可以使用SYSID子句为用户设置一个特定的ID（这类似于一些Unix管理员在操作系统层面上对用户名的处理）。偶尔也要对一个用户进行修改。下面一节解释了如何做到这一点。

8. 创建和修改用户

在这个理论介绍之后，现在是时候实际创建用户，看看事情如何在实际例子中使用。

```
test=# CREATE ROLE bookkeeper NOLOGIN;
CREATE ROLE
test=# CREATE ROLE joe LOGIN;
CREATE ROLE
test=# GRANT bookkeeper TO joe;
GRANT ROLE
```

这里做的第一件事是，创建了一个叫做bookkeeper的角色。

请注意，我们不希望人们以bookkeeper的身份登录，所以该角色被标记为NOLOGIN。

你还应该注意，如果你使用CREATE ROLE子句，NOLOGIN是默认值。如果你喜欢使用CREATE USER子句，默认设置是LOGIN。

然后，joe角色被创建并标记为LOGIN。最后，bookkeeper角色被分配给joe角色，这样他们就可以做bookkeeper实际被允许做的所有事情。

一旦用户到位，我们就可以测试我们目前拥有的东西。

```
[hs@zenbook ~]$ psql test -U bookkeeper
psql: FATAL: role "bookkeeper" is not permitted to log in
```

正如预期的那样，bookkeeper角色不被允许登录系统。如果joe角色试图登录，会发生什么？请看下面的代码片断。

```
[hs@zenbook ~]$ psql test -U joe
...
test=>
```

这实际上将按预期工作。然而，请注意，命令提示符已经改变。这只是PostgreSQL向你显示你没有以超级用户身份登录的一种方式。一旦创建了一个用户，可能有必要对其进行修改。我们可能想改变的一件事是密码。在PostgreSQL中，用户被允许改变他们自己的密码。下面是它的工作原理。

```
test=> ALTER ROLE joe PASSWORD 'abc';
ALTER ROLE
test=> SELECT current_user;
current_user
-----
joe
(1 row)
```

请注意，ALTER ROLE会改变角色的属性。如果已经配置了数据定义语言（DDL）日志，PASSWORD实际上会使密码显示在日志文件中。这不是很理想。最好是用一个可视化工具来改变密码。在这种情况下，有一些协议支持，可以确保密码不会以明文形式在网上发送。直接使用ALTER ROLE来改变密码，根本不是一个好主意。

ALTER ROLE子句（或ALTER USER）将允许我们改变在创建用户时可以设置的大部分设置。然而，管理用户的内容甚至更多。在许多情况下，我们想给一个用户分配特殊的参数。ALTER USER子句为我们提供了这样的方法。

```
ALTER ROLE { role_specification | ALL }
[ IN DATABASE database_name ]
SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER ROLE { role_specification | ALL }
[ IN DATABASE database_name ]
SET configuration_parameter FROM CURRENT
ALTER ROLE { role_specification | ALL }
[ IN DATABASE database_name ] RESET configuration_parameter
ALTER ROLE { role_specification | ALL }
[ IN DATABASE database_name ] RESET ALL
```

语法相当简单，而且相当直截了当。为了说明为什么这真的很有用，我添加了一个真实世界的例子。让我们假设Joe碰巧住在毛里求斯岛。当他登录时，他希望在自己的时区，即使他的数据库服务器位于欧洲。让我们在每个用户的基础上设置时区

```
test=> ALTER ROLE joe SET TimeZone = 'UTC-4';
ALTER ROLE
test=> SELECT now();
now
-----
2020-10-09 20:36:48.571584+01
(1 row)
test=> \q
[hs@zenbook ~]$ psql test -U joe
...
test=> SELECT now();
now
-----
2020-10-09 23:36:53.357845+04
(1 row)
```

ALTER ROLE子句将修改用户。当joe重新连接时，时区将已经为他设置好了。

时区不会被立即改变。你应该重新连接，或者使用SET ... TO DEFAULT子句。

这里重要的是，对于一些内存参数，如work_mem，也可以这样做，这在本书前面已经讲过了

9.定义数据库级安全性

在实例层面上配置完用户后，就可以深入挖掘，看看在数据库层面上可以做什么。出现的第一个主要问题是：我们明确允许joe登录数据库实例，但谁或什么允许joe实际连接到其中一个数据库？也许你不希望joe访问你系统中的所有数据库。限制对某些数据库的访问，正是我们在这个层面上可以实现的。

对于数据库，可以使用GRANT子句来设置以下权限。

```
GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...]
| ALL [ PRIVILEGES ] }
ON DATABASE database_name [, ...]
TO role_specification [, ...] [ WITH GRANT OPTION ]
```

在数据库层面，有两个主要的权限值得密切关注。

- CREATE：这允许某人在数据库中创建一个模式。注意，CREATE子句不允许创建表；它是关于模式的。在PostgreSQL中，表驻留在模式中，所以你必须先进入模式级别，以便能够创建一个表。
- CONNECT（连接）。这允许某人连接到一个数据库

现在的问题是：没有人明确地给joe角色分配任何CONNECT权限，那么这些权限究竟从何而来？答案是这样的：有一个叫做public的东西，它类似于Unix的世界。如果世界被允许做某事，joe也是，他是public的一部分。

最主要的是，public不是一个角色，它可以被放弃和重新命名。我们可以简单地把它看作是系统中每个人的等同物。

因此，为了确保不是每个人都能在任何时候连接到任何数据库，CONNECT可能必须从一般人那里撤销。为此，我们可以以超级用户的身份连接并解决这个问题。

```
[hs@zenbook ~]$ psql test -U postgres
...
test=# REVOKE ALL ON DATABASE test FROM public;
REVOKE
test=# \q
[hs@zenbook ~]$ psql test -U joe
psql: FATAL: permission denied for database "test"
DETAIL: User does not have CONNECT privilege.
```

我们可以看到，joe角色不再被允许连接。在这一点上，只有超级用户可以访问测试

一般来说，在创建其他数据库之前，撤销postgres数据库的权限是个好主意。这个概念背后的想法是，这些权限不会再出现在所有那些新创建的数据库中。如果有人需要访问某个数据库，就必须明确授予这些权限。这些权利不再是自动存在的。

如果我们想允许joe角色连接到测试数据库，请以超级用户身份尝试以下一行。

```
[hs@zenbook ~]$ psql test -U postgres
...
test=# GRANT CONNECT ON DATABASE test TO bookkeeper;
GRANT
test=# \q
[hs@zenbook ~]$ psql test -U joe
...
test=>
```

这里有两个选择：

- 我们可以直接允许joe角色，这样只有joe角色能够连接。

- 或者，我们也可以授予记账员角色权限。记住，joe角色将继承bookkeeper角色的所有权限，所以如果我们希望所有会计都能连接到数据库，给bookkeeper角色分配权限似乎是一个有吸引力的想法。

如果我们给bookkeeper角色授予权限是没有风险的，因为这个角色首先不允许登录到实例，所以它纯粹是作为一个权限的来源。

10. 调整模式层面的权限

一旦我们完成了数据库层面的配置，看一下模式层面是有意义的。

在实际查看模式之前，让我们运行一个小测试。

```
test=> CREATE DATABASE test;
ERROR: permission denied to create database
test=> CREATE USER xy;
ERROR: permission denied to create role
test=> CREATE SCHEMA sales;
ERROR: permission denied for database test
```

我们可以看到，joe今天过得很糟糕，除了连接到数据库外，其他的都不允许。

然而，有一个小小的例外，它让很多人感到惊讶。

```
test=> CREATE TABLE t_broken (id int);
CREATE TABLE
test=> \d
List of relations
Schema | Name | Type | Owner
-----+-----+-----+
public | t_broken | table | joe
(1 rows)
```

默认情况下，public被允许与public模式一起工作，而public模式总是在周围。如果我们对确保我们的数据库安全很感兴趣的话，请确保这个问题得到解决。否则，普通用户将有可能用各种表来干扰你的public模式，整个设置可能会受到影响。你还应该记住，如果有人被允许创建一个对象，这个人也是它的所有者。所有权意味着所有的权限都自动提供给创建者，包括销毁该对象。

要把这些权限从public那里拿走，请以超级用户身份运行下面一行。

```
test=# REVOKE ALL ON SCHEMA public FROM public;
REVOKE
```

从现在起，没有人可以在没有正确权限的情况下把东西放到你的public模式中。下一个列表就是证明。

```
[hs@zenbook ~]$ psql test -U joe
...
test=> CREATE TABLE t_data (id int);
ERROR: no schema has been selected to create in
LINE 1: CREATE TABLE t_data (id int);
```

正如我们所看到的，该命令将失败。这里重要的是将显示的错误信息。PostgreSQL不知道应该把这些表放在哪里。默认情况下，它将尝试把表放到以下模式中的一个。

```
test=> SHOW search_path ;
search_path
-----
"$user", public
(1 row)
```

由于没有叫joe的模式，这不是一个选项，所以PostgreSQL将尝试public模式。由于没有权限，它将抱怨说它不知道把表放在哪里。

如果表格有明确的前缀，情况就会立即改变。

```
test=> CREATE TABLE public.t_data (id int);
ERROR: permission denied for schema public
LINE 1: CREATE TABLE public.t_data (id int);
```

在这种情况下，我们会得到你所期望的错误信息。PostgreSQL拒绝了对public模式的访问。

现在，下一个合乎逻辑的问题是：在模式层面可以设置哪些权限，以便给joe角色更多的权力？让我们来看看。

```
GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
ON SCHEMA schema_name [, ...]
TO role_specification [, ...] [ WITH GRANT OPTION ]
```

CREATE意味着有人可以将对象放入模式中。USAGE意味着有人被允许进入该模式。注意，进入模式并不意味着模式内的东西可以被实际使用；这些权限还没有被定义。这只是意味着用户可以看到这个模式的系统目录。

为了允许joe角色访问它之前创建的表，以下一行将是必要的（以超级用户身份执行）。

```
test=# GRANT USAGE ON SCHEMA public TO bookkeeper;
GRANT
```

joe角色现在能够按预期读取其表：

```
[hs@zenbook ~]$ psql test -U joe
test=> SELECT count(*) FROM t_broken;
count
-----
0
(1 row)
```

joe角色也能够添加和修改行，因为它正好是表的所有者。然而，尽管它已经可以做很多事情，但joe角色还不是万能的。请看下面的语句

```
test=> ALTER TABLE t_broken RENAME TO t_useful;
ERROR: permission denied for schema public
```

让我们仔细看一下实际的错误信息。我们可以看到，该消息抱怨的是模式上的权限，而不是表本身的权限（记住，joe角色拥有该表）。要解决这个问题，必须在模式层面而不是表层面上解决。以超级用户身份运行下面一行。

```
test=# GRANT CREATE ON SCHEMA public TO bookkeeper;
GRANT
```

public模式上的CREATE权限被分配给bookkeeper。

joe角色现在可以把它的表的名字改成一个更有用的名字。

```
[hs@zenbook ~]$ psql test -U joe
test=> ALTER TABLE t_broken RENAME TO t_useful;
ALTER TABLE
```

请记住，如果使用DDL，这是必要的。在我作为PostgreSQL支持服务提供者的日常工作中，我已经看到了几个问题，这变成了一个问题。

11. 使用表格

在处理了绑定地址、网络认证、用户、数据库和模式之后，我们终于来到了表的级别。下面的代码片断显示了可以为一个表设置哪些权限。

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE
| REFERENCES | TRIGGER }
[, ...] | ALL [ PRIVILEGES ] }
ON { [ TABLE ] table_name [, ...]
| ALL TABLES IN SCHEMA schema_name [, ...] }
TO role_specification [, ...] [ WITH GRANT OPTION ]
```

让我来逐一解释这些权限：

- SELECT：这允许你读取一个表。
- INSERT：这允许你向表中添加行（这也包括复制等，它不仅仅是关于INSERT子句）。注意，如果你被允许插入，你不会自动被允许读取。为了能够读取你所插入的数据，需要使用SELECT和INSERT子句。
- UPDATE：这是修改一个表的内容。
- DELETE：这是用来从表中删除记录的。
- TRUNCATE：这允许你使用TRUNCATE子句。注意DELETE和TRUNCATE子句是两个独立的权限，因为TRUNCATE子句会锁定表，而DELETE子句不会这样做（即使没有WHERE条件也不会）。
- REFERENCES：这允许创建外键。在引用列和被引用列上都必须有这个权限，否则，键的创建就不会成功。
- TRIGGER：这允许创建触发器

GRANT子句的好处是，我们可以同时对一个模式中的所有表设置权限。

这大大简化了调整权限的过程。也可以使用WITH GRANT OPTION子句。这个想法是为了确保正常用户可以将权限传递给其他人，这样做的好处是能够相当显著地减少管理员的工作负担。试想一下，一个为数百个用户提供访问的系统。管理所有这些人的工作可能开始变得非常繁重，因此管理员可以指定一些人自己管理数据的一个子集。

12. 处理列级安全

在某些情况下，不是每个人都被允许看到所有的数据。试想一下一家银行。有些人可能会看到一个银行账户的全部信息，而其他人可能只被限制在数据的一个子集。在现实世界中，有人可能不被允许阅读余额栏，而其他人可能看不到人们的贷款利率。

另一个例子是，人们被允许看到其他人的资料，但不允许看到他们的照片或其他一些私人信息。现在的问题是：如何使用列级安全

为了证明这一点，我们将在现有的表中增加一个属于joe角色的列。

```
test=> ALTER TABLE t_useful ADD COLUMN name text;
ALTER TABLE
```

该表现在由两列组成。这个例子的目的是确保用户只能看到其中的一列。

```
test=> \d t_useful
Table "public.t_useful"
Column | Type | Modifiers
-----+-----+
id    | integer |
name  | text  |
```

作为一个超级用户，让我们创建一个用户，并给他们访问包含我们表的模式的权限。

```
test=# CREATE ROLE paul LOGIN;
CREATE ROLE
test=# GRANT CONNECT ON DATABASE test TO paul;
GRANT
test=# GRANT USAGE ON SCHEMA public TO paul;
GRANT
```

不要忘记给新来的人以CONNECT权限，因为在本章的早些时候，CONNECT被撤销了公共的。因此，明确的授予是绝对必要的，以确保我们可以进入到表。

可以给paul角色提供SELECT的权限。下面是它的工作原理。

```
test=# GRANT SELECT (id) ON t_useful TO paul;
GRANT
```

这已经足够了。已经可以以paul用户的身份连接到数据库并读取该列。

```
[hs@zenbook ~]$ psql test -U paul
...
test=> SELECT id FROM t_useful;
id
-----
(0 rows)
```

如果我们使用列级权限，有一件重要的事情要记住。我们应该停止使用SELECT *，因为它将不再起作用。

```
test=> SELECT * FROM t_useful;
ERROR: permission denied for relation t_useful
```

*仍然意味着所有的列，但由于没有办法访问所有的列，事情就会立即出错。

13.配置默认权限

到目前为止，很多东西都已经配置好了。问题是，如果有新的表被添加到系统中会怎样？一个一个地处理这些表，并设置适当的权限，这可能是相当痛苦和冒险的。如果这些事情能自动发生，那不是很好吗？这正是ALTER DEFAULT PRIVILEGES子句的作用。这个想法是给用户一个选项，使PostgreSQL在一个对象出现时自动设置所需的权限。现在不可能简单地忘记设置这些权限了。

下面的列表显示了语法规规范的第一部分。

```
postgres=# \h ALTER DEFAULT PRIVILEGES
Command: ALTER DEFAULT PRIVILEGES
Description: define default access privileges
Syntax:
ALTER DEFAULT PRIVILEGES
[ FOR { ROLE | USER } target_role [, ...] ]
[ IN SCHEMA schema_name [, ...] ]
abbreviated_grant_or_revoke
where abbreviated_grant_or_revoke is one of:
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
[, ...] | ALL [ PRIVILEGES ] }
ON TABLES
TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

该语法的工作方式与GRANT子句相似，因此使用起来很简单、很直观。为了告诉我们它是如何工作的，我编译了一个简单的例子。这个想法是，如果joe角色创建了一个表，paul角色将自动能够使用它。下面的列表显示了如何分配默认权限。

```
test=# ALTER DEFAULT PRIVILEGES FOR ROLE joe IN SCHEMA public GRANT ALL ON
TABLES TO paul;
ALTER DEFAULT PRIVILEGES
```

在我的例子中，公共模式中的所有表都被授予所有权限。

现在让我们以joe角色连接并创建一个表：

```
[hs@zenbook ~]$ psql test -U joe
...
test=> CREATE TABLE t_user (id serial, name text, passwd text);
CREATE TABLE
```

可以看到，现在可以成功创建表了。

以paul角色连接将证明该表已被分配到适当的权限集。

```
[hs@zenbook ~]$ psql test -U paul
...
test=> SELECT * FROM t_user;
 id | name | passwd
----+---+-----+
(0 rows)
```

该表也可以很好地阅读。

14.深入研究 RLS

到目前为止，一个表总是作为一个整体来显示。当表包含100万行时，有可能从中检索出100万行。如果有人有权利阅读一张表，那就是指整个表。在许多情况下，这是不够的。通常情况下，不允许一个用户看到所有的行是可取的。

考虑下面这个现实世界的例子，一个会计正在为许多人做会计工作。包含税率的表格确实应该对每个人都是可见的，因为每个人都必须支付相同的税率。然而，当涉及到实际交易时，会计师可能想确保每个人只被允许看到自己的交易。不应允许A人看到B人的数据。除此之外，允许一个部门的老板看到他们那部分公司的所有数据也是有意义的。

RLS的设计正是为了做到这一点，它使你能够以一种快速和简单的方式建立多租户系统。配置这些权限的方法是制定策略。CREATE POLICY命令为我们提供了一个编写这些规则的方法。让我们来看看CREATE POLICY命令的一个例子。

```
postgres=# \h CREATE POLICY
Command: CREATE POLICY
Description: define a new row level security policy for a table
Syntax:
CREATE POLICY name ON table_name
[ AS { PERMISSIVE | RESTRICTIVE } ]
[ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
[ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
[ USING ( using_expression ) ]
[ WITH CHECK ( check_expression ) ]
URL: https://www.postgresql.org/docs/13/sql-createpolicy.html
```

语法并不难理解。然而，当然要提供一个例子。为了描述如何编写策略，让我们首先以超级用户身份登录，并创建一个包含几个条目的表。

```
test=# CREATE TABLE t_person (gender text, name text);
CREATE TABLE
test=# INSERT INTO t_person
VALUES ('male', 'joe'),
('male', 'paul'),
('female', 'sarah'),
(NULL, 'R2- D2');
INSERT 0 4
```

一个表被创建，数据被成功添加。然后，访问权被授予joe角色，从下面的代码中可以看出。

```
test=# GRANT ALL ON t_person TO joe;
GRANT
```

到目前为止，一切都很正常，由于没有RLS的存在，joe角色实际上可以读取整个表。但是，让我们看看如果为该表启用ROW LEVEL SECURITY会发生什么。

```
test=# ALTER TABLE t_person ENABLE ROW LEVEL SECURITY;
ALTER TABLE
```

有一个拒绝，而且所有的默认策略都到位了，所以joe角色实际上会得到一个空表。

```
test=> SELECT * FROM t_person;
gender | name
-----+-
(0 rows)
```

默认策略有很大的意义，因为用户被迫明确地设置权限。

现在，该表在RLS的控制下，可以以超级用户的身份编写策略。

```
test=# CREATE POLICY joe_pol_1 ON t_person FOR SELECT TO joe USING (gender =
'male');
CREATE POLICY
```

以joe角色登录并选择所有的数据将只返回两行。

```
test=> SELECT * FROM t_person;
gender | name
-----+-
male | joe
male | paul
(2 rows)
```

让我们以更详细的方式检查我们刚刚创建的策略。我们可以看到的第一件事是，该策略实际上有一个名字。它还与一个表相连，并允许进行某些操作（在本例中，是SELECT子句）。然后是USING子句。它定义了允许joe角色查看的内容。因此，USING子句是连接到每个查询的一个强制性过滤器，只选择我们的用户应该看到的行。

还有一个重要的附带说明，如果有不止一个策略，PostgreSQL将使用OR条件。简而言之，更多的策略将使你默认看到更多的数据。在PostgreSQL 9.6中，情况一直是这样的。然而，随着PostgreSQL 10.0的引入，用户可以选择条件是通过OR还是AND的方式连接的

PERMISSIVE | RESTRICTIVE

默认情况下，PostgreSQL是PERMISSIVE的，所以OR连接是在工作。如果我们决定使用RESTRICTIVE，那么这些子句将用AND来连接

现在，假设由于某种原因，已经决定让joe角色也可以看到机器人。要实现我们的目标，有两个选择。第一个选择是简单地使用ALTER POLICY子句来改变现有的策略。

```
postgres=# \h ALTER POLICY
Command: ALTER POLICY
Description: change the definition of a row level security policy
Syntax:
ALTER POLICY name ON table_name RENAME TO new_name
ALTER POLICY name ON table_name
[ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
[ USING ( using_expression ) ]
[ WITH CHECK ( check_expression ) ]
URL: https://www.postgresql.org/docs/13/sql-alterpolicy.html
```

这个列表显示了ALTER POLICY的语法。它与CREATE POLICY相当相似。

第二个选择是创建第二个策略，如下面的例子所示。

```
test=# CREATE POLICY joe_pol_2 ON t_person FOR SELECT TO joe USING (gender IS NULL);
CREATE POLICY
```

创建政策的工作成功了。美中不足的是，除非使用RESTRICTIVE，否则这些策略只是使用前面所说的OR条件连接。因此，PostgreSQL现在将返回三条记录而不是两条。

```
test=> SELECT * FROM t_person;
gender | name
-----+-
male  | joe
male  | paul
| R2-D2
(3 rows)
```

R2-D2的角色现在也包括在结果中，因为它与第二个策略相匹配。

为了描述PostgreSQL是如何运行查询的，我决定包括一个查询的执行计划。

```
test=> explain SELECT * FROM t_person;
QUERY PLAN
-----
Seq Scan on t_person (cost=0.00..21.00 rows=9 width=64)
  Filter: ((gender IS NULL) OR (gender = 'male'::text))
(2 rows)
```

正如我们所看到的，USING子句都被作为强制过滤器添加到查询中。你可能已经注意到在语法定义中，有两种类型的子句。

- USING：该子句过滤已经存在的行。这与SELECT和UPDATE子句等相关。
- CHECK：该子句过滤即将创建的新行，因此它们与INSERT和UPDATE子句有关，以此类推。

如果我们尝试插入一行，会发生以下情况：

```
test=> INSERT INTO t_person VALUES ('male', 'kaarel');
ERROR: new row violates row-level security policy for table "t_person"
```

由于没有INSERT子句的策略，该语句自然会出错。下面是允许插入的策略。

```
test=# CREATE POLICY joe_pol_3 ON t_person FOR INSERT TO joe WITH CHECK (gender IN ('male', 'female'));
CREATE POLICY
```

joe角色被允许在表中添加男性和女性，这在下面的列表中显示。

```
test=> INSERT INTO t_person VALUES ('female', 'maria');
INSERT 0 1
```

但是，也有一个问题。考虑以下示例：

```
test=> INSERT INTO t_person VALUES ('female', 'maria') RETURNING *;
ERROR: new row violates row-level security policy for table "t_person"
```

请记住，只有一个政策是选择男性。这里的麻烦是，该语句将返回一个女性，这是不允许的，因为joe角色是在只选择男性的策略下。

RETURNING * 子句仅适用于男性：

```
test=> INSERT INTO t_person VALUES ('male', 'max') RETURNING *;
gender | name
-----+-
male  | max
(1 row)
INSERT 0 1
```

如果我们不想要这种行为，我们必须写一个真正包含适当的USING条款的策略。RLS是每个数据库侧安全系统中的一个重要组成部分。一旦我们定义了我们的RLS策略，退一步讲，看看你如何检查权限是有意义的。

15. 检查权限

当所有的权限都被设定后，有时需要知道谁有哪些权限。对于管理员来说，找出谁被允许做什么是至关重要的。不幸的是，这个过程并不那么容易，需要一些知识。通常情况下，我是一个命令行使用的忠实粉丝。然而，在权限系统的情况下，使用图形化的用户界面来做事情确实是有意义的。

在我告诉你如何读取PostgreSQL的权限之前，让我们把权限分配给joe角色，这样我们就可以在下一步检查它们。

```
test=# GRANT ALL ON t_person TO joe;
GRANT
```

关于权限的信息可以用psql中的z命令来检索。

```
test=# \x
Expanded display is on.
test=# \z t_person
Access privileges
-[ RECORD 1 ]-----+
-----+
Schema | public
Name  | t_person
Type  | table
Access privileges | postgres=arwdDxt/postgres
+
| joe=arwdDxt/postgres
Column privileges |
Policies | joe_pol_1 (r):
+ | (u): (gender = 'male'::text)
+ | to: joe
+ | joe_pol_2 (r):
+ | (u): (gender IS NULL)
+ | to: joe
+ | joe_pol_3 (a):
+ | (c): (gender = ANY (ARRAY['male'::text, 'female'::text]))
+ | to: joe
```

这将返回所有这些政策，以及关于访问权限的信息。不幸的是，这些快捷方式很难读懂，而且我感觉它们并没有被管理员广泛理解。在这个例子中，joe角色从PostgreSQL中获得了arwdDxt。这些快捷键到底是什么意思？让我们来看看。

- a: 这附加于 INSERT 子句
- r: 读取 SELECT 子句
- w: 这为 UPDATE 子句写入
- d: 这为 DELETE 子句删除
- D: 这用于TRUNCATE子句（引入时，t已经被采用）
- x: 用于引用
- t: 这用于触发器

如果你不知道这段代码，还有第二种方法可以使事情变得更容易阅读。考虑一下下面的函数调用

```
test=# SELECT * FROM aclexplode('{joe=arwdDxt/postgres}');
 grantor | grantee | privilege_type | is_grantable
-----+-----+-----+
 10 | 18481 | INSERT | f
 10 | 18481 | SELECT | f
 10 | 18481 | UPDATE | f
 10 | 18481 | DELETE | f
 10 | 18481 | TRUNCATE | f
 10 | 18481 | REFERENCES | f
 10 | 18481 | TRIGGER | f
(7 rows)
```

正如我们所看到的，权限集以一个简单的表格形式返回，这使工作变得非常简单。对于那些仍然认为在PostgreSQL中检查权限有些麻烦的人，我们Cybertec最近实施了一个额外的解决方案 (<https://www.cybertec-postgresql.com>) :pg_permission。其目的是为你提供简单的视图来更深入地检查安全系统。

要下载pg_permission，请访问我们的GitHub资源库页面：https://github.com/cybertec-postgresql/pg_permission

你可以很容易地克隆版本库，如以下列表所示。

```
git clone https://github.com/cybertec-postgresql/pg_permission.git
Cloning into 'pg_permission'...
remote: Enumerating objects: 111, done.
remote: Total 111 (delta 0), reused 0 (delta 0), pack-reused 111
Receiving objects: 100% (111/111), 33.08 KiB | 292.00 KiB/s, done.
Resolving deltas: 100% (52/52), done.
```

一旦完成，进入该目录，运行make install即可。这两件事已经足够部署扩展了。

```
test=# CREATE EXTENSION pg_permissions;
CREATE EXTENSION
```

pg_permission将部署少量的视图，它们的结构都是相同的。

```
test=# \d all_permissions
```

```

View "public.all_permissions"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
object_type | obj_type | | |
role_name | name | | |
schema_name | name | | |
object_name | text | c | |
column_name | name | | |
permission | perm_type | | |
granted | boolean | | |

Triggers:
permissions_trigger INSTEAD OF UPDATE ON all_permissions
FOR EACH ROW EXECUTE FUNCTION permissions_trigger_func()

```

all_permissions是一个简单的视图，为你提供所有权限的整体视图。你可以简单地按对象类型过滤，等等，来挖掘细节。同样有趣的是，这个视图上有一个触发器，所以如果你想改变权限，你可以简单地在所有这些视图上运行UPDATE，而pg_permissions将为你改变系统的权限。在下一节中，你将学习如何重新分配所有权。

16.重新分配对象和删除用户

在分配权限和限制访问之后，可能会发生用户从系统中被删除的情况。不出所料，这样做的命令是DROP ROLE和DROP USER命令。下面是DROP ROLE的语法。

```

test=# \h DROP ROLE
Command: DROP ROLE
Description: remove a database role
Syntax:
DROP ROLE [ IF EXISTS ] name [, ...]
URL: https://www.postgresql.org/docs/13/sql-droprole.html

```

一旦讨论了DROP ROLE的语法，我们可以试一试。下面的列表显示了它是如何工作的。

```

test=# DROP ROLE joe;
ERROR: role "joe" cannot be dropped because some objects depend on it
DETAIL: target of policy joe_pol_3 on table t_person
target of policy joe_pol_2 on table t_person
target of policy joe_pol_1 on table t_person
privileges for table t_person
owner of table t_user
owner of sequence t_user_id_seq
owner of default privileges on new relations belonging to role joe in schema
public
owner of table t_useful

```

PostgreSQL会发出错误信息，因为只有当一个用户的所有东西都被夺走了，才能将其删除。这是有道理的，原因如下：只是假设有人拥有一个表。PostgreSQL应该如何处理这个表呢？总得有人拥有它。

要把表从一个用户重新分配给下一个用户，可以考虑看一下REASSIGN子句。

```
test=# \h REASSIGN
Command: REASSIGN OWNED
Description: change the ownership of database objects owned by a database role
Syntax:
REASSIGN OWNED BY { old_role | CURRENT_USER | SESSION_USER } [, ...]
    TO { new_role | CURRENT_USER | SESSION_USER }
URL: https://www.postgresql.org/docs/13/sql-reassign-owned.html
```

语法也很简单，有助于简化切换过程。下面是一个例子。

```
test=# REASSIGN OWNED BY joe TO postgres;
REASSIGN OWNED
```

那么，让我们再次尝试删除 joe 角色：

```
test=# DROP ROLE joe;
ERROR: role "joe" cannot be dropped because some objects depend on it
DETAIL: target of policy joe_pol_3 on table t_person target of policy joe_pol_2
on table t_person
target of policy joe_pol_1 on table t_person privileges for table t_person
owner of default privileges on new relations belonging to role joe in schema
public
```

正如我们所看到的，问题的清单已经大大减少。我们现在能做的是一个接一个地解决所有这些问题，然后放弃这个角色。据我所知，没有任何捷径。使之更有效率的唯一方法是确保尽可能少的权限被分配给真实的人。试着把尽可能多的权限抽象成角色，而这些角色又可以被很多人使用。如果个别权限不分配给真实的人，一般来说，事情就会变得容易。

17.总结

数据库安全是一个广泛的领域，一个30页的章节很难涵盖PostgreSQL安全的所有方面。很多东西，比如 SELinux和SECURITY DEFINER/INVOKER，都没有被触及。然而，在这一章中，我们学到了作为 PostgreSQL开发者和数据库管理员所要面对的最常见的东西。我们还学习了如何避免基本的陷阱，以及如何使我们的系统更加安全。

在第9章，处理备份和恢复，我们将学习PostgreSQL流式复制和增量备份。本章还将介绍故障转移的情况。

18.问题

- 如何配置对PostgreSQL的网络访问？
- 什么是用户，什么是角色？
- 如何改变密码？
- 什么是RLS？

这些问题的答案可以在GitHub仓库中找到 (<https://github.com/PacktPublishing/Mastering-PostgreSQL-13-Fourth-Edition>) 。

处理备份和恢复

1 执行简单的转储

- 1.1 运行pg_dump
- 1.2 传递密码和连接信息
 - 1.2.1 使用环境变量
 - 1.2.2 使用.pgpass
 - 1.2.3 使用服务文件
 - 1.2.4 提取数据子集

2 处理各种格式

3. 重放备份

4. 处理全局数据

5. 总结

6. 问题

在第8章 "管理PostgreSQL的安全" 中，我们看了关于以最简单和最有利的方式保护PostgreSQL所需要知道的一切。本章要讲的主题是备份和恢复。备份应该是一项常规工作，每个管理员都应该关注这项重要的工作。幸运的是，PostgreSQL提供了创建备份的简单方法。

因此，在本章中，我们将涵盖以下主题：

- 执行简单的转储
- 处理各种格式
- 重放备份
- 处理全局数据

在本章结束时，你将能够建立适当的备份机制。

1 执行简单的转储

备份和数据导出是很重要的。如果你正在运行一个PostgreSQL设置，基本上有两种主要的方法来执行备份。

- 逻辑转储（提取代表你的数据的SQL脚本）
- 事务日志传输

事务日志运输背后的想法是对数据库的二进制变化进行存档。大多数人声称，事务日志运输是创建备份的唯一真正方法。然而，在我看来，这不一定是真的。

许多人依靠pg_dump来简单地提取数据的文本表示。有趣的是，pg_dump也是最古老的创建备份的方法，在PostgreSQL项目的早期就已经存在了（事务日志运输是后来加入的）。每个PostgreSQL管理员迟早都会熟悉pg_dump，所以了解它的真正工作原理和作用是很重要的。

1.1 运行pg_dump

在本节的第一部分，你将学习关于pg_dump的一些基本内容。我们要做的第一件事是创建一个简单的文本转储。

```
[hs@linuxpc ~]$ pg_dump test > /tmp/dump.sql
```

这是你能想象到的最简单的备份。基本上，pg_dump登录到本地数据库实例连接到一个叫做test的数据库，并开始提取所有的数据，然后将其发送到stdout并重定向到文件中。这里的好处是，标准输出给了你Unix系统的所有灵活性。你可以很容易地用管道来压缩数据，或者做任何你想做的事情。

在某些情况下，你可能想以一个不同的用户来运行pg_dump。所有的PostgreSQL客户端程序都支持一组一致的命令行参数来传递用户信息。如果你只是想设置用户，使用-U标志，如下所示。

```
[hs@linuxpc ~]$ pg_dump -U whatever_powerful_user test > /tmp/dump.sql
```

下面这组参数可以在所有PostgreSQL客户端程序中找到。

```
...
Connection options:
-d, --dbname=DBNAME database to dump
-h, --host=HOSTNAME database server host or
socket directory
-p, --port=PORT database server port number
-U, --username=NAME connect as specified database user
-w, --no-password never prompt for password
-w, --password force password prompt (should
happen automatically)
--role=ROLENAME do SET ROLE before dump
...
```

你只要把你想要的信息传递给pg_dump，如果你有足够的权限，PostgreSQL就会获取数据。这里重要的是看这个程序到底是如何工作的。基本上，pg_dump连接到数据库并打开一个大型的可重复读取事务，简单地读取所有的数据。记住，可重复读取确保PostgreSQL创建一个一致的数据快照，这个快照在整个事务中不会改变。换句话说，转储总是一致的--没有外键会被违反。输出是转储开始时的数据快照。一致性是这里的一个关键因素。这也意味着，在转储运行时对数据所做的改变将不会再出现在备份中。

转储只是读取所有的东西--因此，没有单独的权限可以转储东西。只要你能读取它，你就能备份它。

另外，请注意，备份默认为文本格式。这意味着你可以安全地从比如说Solaris中提取数据，并将其转移到其他一些CPU架构上。在二进制拷贝的情况下，这显然是不可能的，因为磁盘上的格式取决于你的CPU架构。

1.2 传递密码和连接信息

如果你仔细看一下上一节中显示的连接参数，你会注意到没有办法向pg_dump传递密码。你可以强制执行密码提示，但你不能用命令行选项把参数传给pg_dump。

原因很简单，因为密码可能会出现在进程表中，并被其他人看到。现在的问题是：如果服务器上的pg_hba.conf强制执行密码，客户程序如何提供密码？

有各种手段可以做到这一点。这里有三种：

- 使用环境变量
- 使用.pgpass
- 使用服务文件

在本节中，我们将了解所有三种方法。

1.2.1 使用环境变量

传递各种参数的方法之一是使用环境变量。如果信息没有明确地传递给pg_dump，它将在预定义的环境变量中寻找缺少的信息。所有潜在设置的列表可以在<https://www.postgresql.org/docs/13/static/libpq-envars.html>找到。

下面的概述显示了一些备份时通常需要的环境变量。

- PGHOST: 这告诉系统要连接到哪个主机。
- PGPORT: 这定义了要使用的TCP端口。
- PGUSER: 这告诉客户程序所需的用户。
- PGPASSWORD: 这包含要使用的密码。
- PGDATABASE: 这是要连接的数据库的名称。

这些环境的优点是密码不会出现在进程表中。然而，还有更多。考虑以下示例：

```
psql -U ... -h ... -p ... -d ...
```

考虑到你是一个系统管理员，你真的想每天都打几遍这样的长代码吗？如果你反复使用同一台主机，只需设置这些环境变量并使用普通SQL进行连接。下面的列表显示了如何连接：

```
[hs@linuxpc ~]$ export PGHOST=localhost
[hs@linuxpc ~]$ export PGUSER=hs
[hs@linuxpc ~]$ export PGPASSWORD=abc
[hs@linuxpc ~]$ export PGPORT=5432
[hs@linuxpc ~]$ export PGDATABASE=test
[hs@linuxpc ~]$ psql
psql (13.0)
Type "help" for help.
```

正如你所看到的，不再有任何命令行参数。只要输入psql就可以了。

所有基于标准PostgreSQL C语言客户端库（libpq）的应用程序都会理解这些环境变量，因此你不仅可以在psql和pg_dump中使用它们，还可以在许多其他应用程序中使用。

1.2.2 使用.pgpass

一个非常常见的存储登录信息的方法是通过使用.pgpass文件。这个想法很简单：把一个叫做.pgpass的文件放到你的主目录中，把你的登录信息放在那里。其格式很简单。下面的列表包含了基本的格式

```
hostname:port:database:username:password
```

一个例子如下：

```
192.168.0.45:5432:mydb:xy:abc
```

PostgreSQL提供了一些很好的附加功能，其中大多数字段都可以包含*。下面是一个例子。

```
*:*:*:xy:abc
```

这个*字符意味着在每个主机、每个端口、每个数据库中，名为xy的用户将使用abc作为密码。要使PostgreSQL使用.pgpass文件，确保有正确的文件权限。如果没有以下几行，事情就不能正常进行。

```
chmod 0600 ~/ .pgpass
```

chmod设置文件级权限。这对于保护文件是必要的。此外，.pgpass也可以在Windows系统上使用。在这种情况下，该文件可以在%APPDATA%\postgresql\pgpass.conf路径下找到。

1.2.3 使用服务文件

然而，.pgpass并不是你可以使用的唯一文件。你还可以利用服务文件。它是这样工作的：如果你想一次又一次地连接到相同的服务器，你可以创建一个.pg_service.conf文件。它将保存你需要的所有连接信息。

以下是.pg_service.conf文件的示例：

```
Mac:~ hs$ cat .pg_service.conf
# a sample service
[hansservice]
host=localhost
port=5432
dbname=test
user=hs
password=abc
[paulservice]
host=192.168.0.45
port=5432
dbname=xyz
user=paul
password=cde
```

要连接到其中一个服务，只需设置环境并连接：

```
iMac:~ hs$ export PGSERVICE=hansservice
```

现在可以在不向psql传递参数的情况下建立一个连接。

```
iMac:~ hs$ psql
psql (13.0)
Type "help" for help.
test=#
```

正如你所看到的，登录工作无需额外的命令行参数。另外，你也可以使用以下命令。

```
psql service=hansservice
```

现在我们已经学会了如何传递密码和连接信息，让我们继续学习如何提取数据的子集

1.2.4 提取数据子集

到目前为止，我们已经看到如何转储整个数据库。然而，这可能不是我们想要做的事情。在许多情况下，我们只想提取一个表或模式的子集。幸运的是，`pg_dump`可以帮助我们做到这一点，同时还提供了几个开关。

- `-a`: 这只转储数据，不转储数据结构。
- `-s`: 这将转储数据结构，但跳过数据。
- `-n`: 只转储特定的模式。
- `-N`: 转储所有数据，但不包括某些模式。
- `-t`: 只转储某些表。
- `-T`: 转储所有数据，但不包括某些表（如果你想排除日志表等，这是有意义的）。

部分转储可以非常有用，可以大大加快事情的进展。现在我们已经学会了如何执行简单的转储，让我们学习如何处理各种文件格式。

2 处理各种格式

到目前为止，我们已经看到`pg_dump`可以用来创建文本文件。这里的问题是，一个文本文件只能被完全重放。如果我们保存了整个数据库，我们只能重放整个数据库。在大多数情况下，这不是我们想要的。因此，PostgreSQL有额外的格式，提供更多的功能。

至此，支持四种格式：

```
-F, --format=c|d|t|p output file format (custom, directory, tar, plain text  
(default))
```

我们已经看到了纯文本，这只是普通的文本。在此基础上，我们可以使用自定义格式。自定义格式背后的想法是要有一个压缩的转储，包括一个目录。这里有两种创建自定义格式转储的方法。

```
[hs@linuxpc ~]$ pg_dump -Fc test > /tmp/dump.fc  
[hs@linuxpc ~]$ pg_dump -Fc test -f /tmp/dump.fc
```

除了目录之外，压缩转储还有一个好处：它的体积要小得多。经验法则是，一个自定义格式的转储比你要备份的数据库实例小90%左右。当然，这在很大程度上取决于索引的数量，但是对于许多数据库应用来说，这个粗略的估计是正确的。

创建备份后，我们可以检查备份文件：

```
[hs@linuxpc ~]$ pg_restore --list /tmp/dump.fc  
;  
;  
; Archive created at 2020-09-15 10:26:46 UTC  
;  
; dbname: test  
;  
; TOC Entries: 25  
;  
; Compression: -1  
;  
; Dump Version: 1.14-0  
;  
; Format: CUSTOM  
;  
; Integer: 4 bytes  
;  
; Offset: 8 bytes  
;  
; Dumped from database version: 13.0  
;  
; Dumped by pg_dump version: 13.0  
;  
;
```

```
; Selected TOC Entries:  
;  
3103; 1262 16384 DATABASE - test hs  
3; 2615 2200 SCHEMA - public hs  
3104; 0 0 COMMENT - SCHEMA public hs  
1; 3079 13350 EXTENSION - plpgsql  
3105; 0 0 COMMENT - EXTENSION plpgsql  
187; 1259 16391 TABLE public t_test hs  
...
```

注意，`pg_restore --list`将返回备份的目录。

使用自定义格式是一个好主意，因为备份的大小会缩小。然而，还有一点：`-Fd`命令将以目录格式创建备份。现在你将得到一个包含几个文件的目录，而不是一个单一的文件。

```
[hs@linuxpc ~]$ mkdir /tmp/backup  
[hs@linuxpc ~]$ pg_dump -Fd test -f /tmp/backup/  
[hs@linuxpc ~]$ cd /tmp/backup/  
[hs@linuxpc backup]$ ls -lh  
total 86M  
-rw-rw-r--. 1 hs hs 85M Jan 4 15:54 3095.dat.gz  
-rw-rw-r--. 1 hs hs 107 Jan 4 15:54 3096.dat.gz  
-rw-rw-r--. 1 hs hs 740K Jan 4 15:54 3097.dat.gz  
-rw-rw-r--. 1 hs hs 39 Jan 4 15:54 3098.dat.gz  
-rw-rw-r--. 1 hs hs 4.3K Jan 4 15:54 toc.dat
```

目录格式的一个优点是我们可以使用一个以上的内核来执行备份。在普通格式或自定义格式的情况下，只有一个进程会被`pg_dump`使用。而目录格式则改变了这一规则。下面的例子显示了我们如何告诉`pg_dump`使用四个内核（作业）。

```
[hs@linuxpc backup]$ rm -rf *  
[hs@linuxpc backup]$ pg_dump -Fd test -f /tmp/backup/ -j 4
```

在本节中，你已经了解了基本的文本转储。在下一节中，你将学习备份回放。

我们数据库中的对象越多，潜在加速的可能性就越大。

3. 重放备份

除非你尝试过实际重放，否则拥有备份是毫无意义的。幸运的是，这很容易做到。如果你已经创建了一个明文备份，只需拿起SQL文件并执行它。下面的例子显示了如何做到这一点。

```
psql your_db < your_file.sql
```

一个明文备份只是一个包含所有内容的文本文件。我们总是可以简单地重放一个文本文件。

如果你已经决定采用自定义格式或目录格式，你可以使用`pg_restore`来重放备份。此外，`pg_restore`允许你做各种花哨的事情，比如只重放数据库的一部分。然而，在大多数情况下，你将简单地重放整个数据库。在这个例子中，我们将创建一个空的数据库，只是重放一个自定义格式的转储。

```
[hs@linuxpc backup]$ createdb new_db  
[hs@linuxpc backup]$ pg_restore -d new_db -j 4 /tmp/dump.fc
```

请注意，pg_restore将把数据添加到一个现有的数据库中。如果你的数据库不是空的，pg_restore可能会出错，但会继续。

同样，-j被用来抛出一个以上的进程。在这个例子中，四个核心被用来重放数据；然而，这只有在重放一个以上的表时才有效。

如果你使用的是目录格式，你可以简单地传递目录的名称而不是文件。

就性能而言，如果你正在处理少量或中等数量的数据，转储是一个好的解决方案。有两个主要的缺点。

- 我们将得到一个快照，所以自上次快照以来的一切都将丢失。
- 与二进制副本相比，从头开始重建转储的速度相对较慢，因为所有的索引都要重建。

我们将在第10章 "备份的意义" 和 "复制" 中对二进制备份进行考察。复制备份是很容易的，但是还有更多的东西没有被发现。下面的章节将处理全局数据。那是什么意思？

4. 处理全局数据

在前面的章节中，我们了解了pg_dump和pg_restore，它们是创建备份时的两个重要程序。问题是，pg_dump创建数据库转储--它在数据库层面上工作。如果我们想备份整个实例，我们必须使用pg_dumpall或者分别转储所有的数据库。在我们讨论这个问题之前，先看看pg_dumpall是如何工作的，这很有意义。

```
pg_dumpall > /tmp/all.sql
```

让我们看看：pg_dumpall将连接到一个又一个的数据库，并将东西发送到stdout，在那里你可以用Unix处理它。注意，pg_dumpall可以像pg_dump一样使用。然而，它有一些缺点。它不支持自定义或目录格式，因此不提供多核支持。这意味着我们将只能用一个线程。

然而，pg_dumpall还有很多内容。请记住，用户是生活在实例级别的。如果你创建一个普通的数据库转储，你会得到所有的权限，但是你不会得到所有的CREATE USER语句。全局变量不包含在普通转储中--它们只会被pg_dumpall提取出来。

如果我们只想要全局变量，我们可以使用-g选项运行pg_dumpall：

```
pg_dumpall -g > /tmp/globals.sql
```

在大多数情况下，你可能想运行pg_dumpall -g，以及一个自定义或目录格式的转储来提取你的实例。一个简单的备份脚本可能看起来像这样。

```
#!/bin/sh  
  
BACKUP_DIR=/tmp/  
  
pg_dumpall -g > $BACKUP_DIR/globals.sql  
  
for x in $(psql -c "SELECT datname FROM pg_database  
WHERE datname NOT IN ('postgres', 'template0', 'template1')" postgres -A -t)  
do  
pg_dump -Fc $x > $BACKUP_DIR/$x.fc done
```

它将首先转储全局变量，然后循环遍历数据库列表，以自定义格式逐个提取它们。

5.总结

在这一章中，我们了解了创建备份和转储的一般情况。到目前为止，还没有涉及二进制备份，但你已经能够从服务器中提取文本备份，这样你就能以最简单的方式保存和重放你的数据。数据保护很重要，而备份对于确保数据安全至关重要。

在第10章 "理解备份和复制" 中，你将学习事务日志运输、流式复制和二进制备份的知识。你还将学习如何使用PostgreSQL的内置工具来复制实例。

6.问题

- 每个人都应该创造转储吗？
- 为什么转储量这么小？
- 你也必须转储全局变量吗？
- 有一个.pgpass文件是安全的吗？

这些问题的答案可以在GitHub仓库中找到 (<https://github.com/PacktPublishing/Mastering-PostgreSQL-13-Fourth-Edition>) 。

理解备份和复制

1.了解事务日志

1.1 查看事务日志

1.2 了解检查点

1.3 优化事务日志

2.事务日志的存档和恢复

2.1 配置归档

2.2 配置 pg_hba.conf 文件

2.3 创建基础备份

2.4 减少备份带宽

2.5 映射表空间

2.6 使用不同的格式

2.7 测试事务日志归档

2.8 重放事务日志

2.9 找到正确的时间戳

2.10 清理事务日志存档

3.设置异步复制

3.1 执行基本设置

3.2 提高安全性

3.3 停止和恢复复制

3.4 检查复制以确保可用性

3.5 执行故障转移和了解时间表

3.6 管理冲突

3.7 使复制更可靠

4.升级到同步复制

4.1 调整耐久度

5.利用复制槽

5.1 处理物理复制槽

5.2 处理逻辑复制槽

5.3 逻辑复制槽的用例

6.利用CREATE PUBLICATION和CREATE SUBSCRIPTION命令

7.总结

8.问题

在第9章 "处理备份和恢复" 中，我们学到了很多关于备份和恢复的知识，这对管理来说至关重要。到目前为止，只涉及到了逻辑备份；我将在本章中改变这种情况。

这一章是关于PostgreSQL的事务日志，以及我们可以用它来改善我们的设置，使事情更安全。

在本章中，我们将讨论以下主题：

- 了解事务日志
- 事务日志的存档和恢复
- 设置异步复制
- 升级到同步复制
- 利用复制槽
- 利用CREATE PUBLICATION和CREATE SUBSCRIPTION命令

在本章结束时，你将能够设置事务日志存档和复制。请记住，本章不可能成为复制的全面指南；它只是一个简短的介绍。对复制的全面介绍需要500页左右。只是作为比较，仅PostgreSQL Replication一书，也是来自Packt，就接近400页。本章将以更紧凑的形式涵盖最基本的东西。

1.了解事务日志

每一个现代的数据库系统都提供了一些功能，以确保系统能够在出错或有人拔掉插头的情况下经受住崩溃。这对文件系统和数据库系统都是如此。PostgreSQL也提供了一种方法来确保崩溃不能损害数据的完整性或数据本身。它可以保证，如果断电，系统总是能够再次启动并完成其工作。

提供这种安全的手段是通过提前写入日志（WAL）或xlog实现的。这个想法是不直接写进数据文件，而是先写进日志。为什么这很重要？想象一下，我们正在写一些数据，如下所示。

```
INSERT INTO data ... VALUES ('12345678');
```

我们假设这些数据是直接写到数据文件中的。如果操作中途失败，数据文件就会被破坏。它可能包含写了一半的行，没有索引指针的列，丢失的提交信息，等等。由于硬件并不能真正保证大块数据的原子写入，所以必须找到一种方法来使其更加健壮。通过写到日志而不是直接写到文件，这个问题可以得到解决。

在 PostgreSQL 中，事务日志由记录组成。

一个单一的写可以由各种记录组成，这些记录都有一个校验和，并被链在一起。一个单一的事务可能包含B树、索引、存储管理器、提交记录，以及更多。每种类型的对象都有自己的WAL条目，以确保该对象能够在崩溃后存活。如果发生崩溃，PostgreSQL将启动并根据事务日志修复数据文件，以确保不允许发生永久性损坏。

介绍完了，现在让我们对事务日志有一个基本的了解。

1.1 查看事务日志

在PostgreSQL中，WAL通常可以在数据目录下的pg_wal目录中找到，除非在initdb中另外指定。在PostgreSQL的旧版本中，WAL目录被称为pg_xlog，但随着PostgreSQL 10.0的引入，该目录被重新命名。

其原因是，更多的时候，人们会删除pg_xlog目录的内容，这当然会导致严重的问题和潜在的数据库损坏。因此，社区采取了史无前例的措施，对PostgreSQL实例内的目录进行重命名。希望能使这个名字足够可怕，以至于没有人敢于再次删除内容。

以下清单显示了 pg_wal 目录的样子：

```
[postgres@zenbook pg_wal]$ pwd  
/var/lib/pgsql/13/data/pg_wal  
[postgres@zenbook pg_wal]$ ls -l  
total 688132  
-rw----- 1 postgres postgres 16777216 Jan 19 07:58 000000010000000000000000CD  
-rw----- 1 postgres postgres 16777216 Jan 13 17:04 000000010000000000000000CE  
-rw----- 1 postgres postgres 16777216 Jan 13 17:04 000000010000000000000000CF  
-rw----- 1 postgres postgres 16777216 Jan 13 17:04 000000010000000000000000D0  
-rw----- 1 postgres postgres 16777216 Jan 13 17:04 000000010000000000000000D1  
-rw----- 1 postgres postgres 16777216 Jan 13 17:04 000000010000000000000000D2
```

我们可以看到的是，事务日志是一个16MB的文件，由24位数字组成。编号是十六进制的。我们可以看到，CF后面是D0。这些文件总是一个固定的大小。

有一点需要注意的是，在PostgreSQL中，事务日志文件的数量与事务的大小无关。你可以有一组非常小的事务日志文件，但仍然可以轻松地运行一个多TB级的事务。

传统上，WAL目录通常由16MB的文件组成。然而，自从引入PostgreSQL后，现在可以用initdb设置WAL段的大小。在某些情况下，这可以加快事情的进展。下面是它的工作原理。下面的例子告诉我们如何将WAL文件的大小改为32MB。

```
initdb -D /pgdata --wal-segsize=32
```

1.2 了解检查点

正如我前面提到的，每一个变化都是以二进制格式写入WAL的（它不包含SQL）。问题是这样的--数据库服务器不能永远向WAL写下去，因为随着时间的推移，它将消耗越来越多的空间。所以，在某些时候，事务日志必须被回收。这是由检查点完成的，它在后台自动发生。

这个想法是，当数据被写入时，它首先进入事务日志，然后一个脏缓冲区被放入共享缓冲区。这些脏缓冲区必须进入磁盘，由后台写入器或在检查点期间写出到数据文件中。一旦所有的脏缓冲区都被写入，事务日志就可以被删除。

请永远不要手动删除事务日志文件。在崩溃的情况下，数据库服务器将无法再次启动，而且随着新事务的到来，所需的磁盘空间量无论如何都会被回收。永远不要手动触摸事务日志。PostgreSQL会自己处理事情，在那里做事情真的很有害。

1.3 优化事务日志

检查点是自动发生的，由服务器触发。然而，有一些配置设置决定何时启动检查点。postgresql.conf文件中的下列参数负责处理检查点。

```
#checkpoint_timeout = 5min # range 30s-1d  
#max_wal_size = 1GB  
#min_wal_size = 80MB
```

启动检查点有两个原因：

- 如果我们可能耗尽了时间或空间。
- 两个检查点之间的最大时间由checkpoint_timeout变量定义。

为存储事务日志提供的空间量将在min_wal_size和max_wal_size变量之间变化。PostgreSQL会自动触发检查点，真正需要的空间量将介于这两个数字之间。

max_wal_size变量是一个软限制，PostgreSQL可能（在重负载下）暂时需要多一点空间。换句话说，如果我们的事务日志是在一个单独的磁盘上，确保实际上有多一点的空间可以用来存储WAL是有意义的。

有人如何调整PostgreSQL 9.6和13.0中的事务日志？在9.6中，对后台写入器和检查点机制做了一些改变。在旧版本中，有一些用例，从性能的角度来看，较小的检查点距离实际上是有意义的。在9.6及以后的版本中，这种情况已经基本改变，更宽的检查点距离基本上总是非常有利的，因为许多优化可以在数据库和操作系统层面上应用，以加快事情。最值得注意的优化是，块在被写出之前被排序，这大大减少了机械磁盘上的随机I/O。

但还有一点。大的检查点距离实际上会减少创建的WAL的数量。是的，这是正确的 - 较大的检查点距离将导致更少的WAL。

这样做的原因很简单。每当一个区块在检查点之后第一次被触及，它就必须被完全发送到WAL。如果区块被更频繁的改变，只有改变的部分才会被送到日志中。较大的距离基本上会导致较少的全页写入，这反过来又减少了首先创建的WAL的数量。这种差异可能是相当大的，正如在我的一篇博文中所看到的：<https://www.postgresql-support.com/checkpoint-distance-and-amount-of-wal/>。

PostgreSQL还允许我们配置检查点是否应该短而密集，或者是否应该分散在较长的时间内。默认值是0.5，这意味着检查点的方式应该是在当前检查点和下一个检查点之间，进程已经完成一半。下面的列表显示了checkpoint_completion_target。

```
#checkpoint_completion_target = 0.5
```

增加这个值基本上意味着检查点被拉长，强度降低。在许多情况下，一个较高的值已被证明有利于平缓由密集检查点引起的I/O峰值。

然而，性能并不是唯一重要的问题。让我们也来看看日志归档和恢复的问题。

2.事务日志的存档和恢复

在我们简单介绍了事务日志的总体情况后，现在是时候关注事务日志的归档过程了。正如我们已经看到的，事务日志包含了对存储系统所做的二进制变化的序列。那么，为什么不使用它来复制数据库实例，并做很多其他很酷的事情，比如归档？

2.1 配置归档

在本章中，我们要做的第一件事是创建一个配置来执行标准的时间点恢复（PITR）。与普通转储相比，使用PITR有几个优点。

- 我们将损失更少的数据，因为我们可以将数据恢复到某个时间点，而不是仅仅恢复到固定的备份点。
- 恢复的速度会更快，因为索引不需要从头开始创建。它们只是被复制过来，并且可以随时使用。

PITR的配置很简单。只需在postgresql.conf文件中做一些修改，如下表中所示。

```
wal_level = replica # used to be "hot_standby" in older versions  
max_wal_senders = 10 # at least 2, better at least 2
```

wal_level变量表示服务器应该产生足够的事务日志，以允许进行PITR。如果wal_level变量被设置为最小值（这是到PostgreSQL 9.6为止的默认值），事务日志将只包含足够的信息来恢复单节点设置--它不够丰富，无法处理复制。在PostgreSQL 10.0中，默认值已经正确，不再需要改变大多数设置。

max_wal_senders变量将允许我们从服务器上流传WAL。它将允许我们使用pg_basebackup来创建一个初始备份，而不是传统的基于文件的复制。这里的好处是pg_basebackup更容易使用。同样，10.0中的默认值已经被改变，对于90%的设置，不需要改变。

WAL流背后的想法是，创建的事务日志被复制到一个安全的地方进行存储。基本上，有两种传输WAL的手段。

- 使用pg_receivewal（到9.6为止，这被称为pg_receivexlog）
- 使用文件系统作为存档手段

在本节中，我们将看看如何设置第二个选项。在正常的操作中，PostgreSQL会不断向这些WAL文件写东西。当我们在postgresql.conf文件中设置archive_mode = on时，PostgreSQL将为每一个文件调用archive_command变量。

一个配置可能看起来如下。首先，可以创建一个存储这些交易日志文件的目录。

```
mkdir /archive  
chown postgres.postgres archive
```

可以在postgresql.conf文件中修改以下条目。

```
archive_mode = on  
archive_command = 'cp %p /archive/%f'
```

重启就可以实现归档，但让我们先配置pg_hba.conf文件，把停机时间降到绝对最低。

注意，我们可以把任何命令放入archive_command变量中。

许多人使用rsync、scp和其他方式将他们的WAL文件传送到一个安全的地方。如果我们的脚本返回0，PostgreSQL会认为该文件已经被归档。如果返回的是其他信息，PostgreSQL将尝试再次归档该文件。这是必要的，因为数据库引擎必须确保没有文件丢失。为了执行恢复过程，我们必须要有每一个文件可用；不允许有一个文件丢失。在下一步，我们将调整pg_hba.conf文件中的配置

2.2 配置 pg_hba.conf 文件

现在postgresql.conf文件已经配置成功，有必要对pg_hba.conf文件进行流式配置。注意，只有当我们计划使用pg_basebackup时才有必要这样做，它是创建基础备份的最先进的工具。

基本上，我们在pg_hba.conf文件中的选项与我们在第8章管理PostgreSQL安全中已经看到的选项相同。只有一个主要问题需要记住，可以借助下面的代码来理解。

```
# Allow replication connections from localhost, by a user with the  
# replication privilege.  
local replication postgres trust  
host replication postgres 127.0.0.1/32 trust  
host replication postgres ::1/128 trust
```

我们可以定义标准的pg_hba.conf文件规则。重要的是，第二列说的是复制。普通的规则是不够的--添加明确的复制权限真的很重要。另外，请记住，我们不一定要以超级用户的身份做这件事。我们可以创建一个特定的用户，只允许他进行登录和复制

同样，PostgreSQL 10及以后的版本已经按照我们在本节中所概述的方式进行了配置。当开箱即用的远程IP必须被添加到pg_hba.conf中时，本地复制就可以工作了

现在pg_hba.conf文件已经被正确配置，可以重新启动PostgreSQL。

2.3 创建基础备份

在教会PostgreSQL如何归档这些WAL文件之后，是时候创建第一个备份了。我们的想法是要有一个备份，并根据该备份重放WAL文件，以达到任何时间点。

为了创建一个初始备份，我们可以求助于pg_basebackup，它是一个用于执行备份的命令行工具。让我们调用pg_basebackup，看看它是如何工作的。

```
pg_basebackup -D /some_target_dir  
-h localhost  
--checkpoint=fast  
--wal-method=stream
```

如我们所见，我们将在这里使用四个参数：

- -D: 我们要把基本备份放在哪里? PostgreSQL需要一个空目录。在备份结束时, 我们将看到服务器的数据目录(目标)的副本。
- -h: 这表示主服务器(源)的IP地址或名称。这是你要备份的服务器。
- --checkpoint=fast。通常情况下, pg_basebackup会等待主服务器创建一个检查点。这样做的原因是, 重放过程必须从某个地方开始。一个检查点可以确保数据已经写到某一点, 所以PostgreSQL可以安全地跳到那里, 开始重放过程。基本上, 不使用--checkpoint=fast参数也可以做到这一点。然而, 在这种情况下, pg_basebackup可能需要一段时间才能开始复制数据。检查点的间隔可以达到1小时, 这可能会不必要地拖延我们的备份。
- --wal-method=stream。默认情况下, pg_basebackup会连接到主服务器并开始复制文件过来。现在, 请记住, 这些文件在复制的过程中会被修改。因此, 到达备份的数据是不一致的。这种不一致可以在恢复过程中使用WAL进行修复。然而, 备份本身是不一致的。通过添加-wal-method=stream参数, 可以创建一个独立的备份; 它可以直接启动, 而不需要重放事务日志。如果我们只想克隆一个实例而不使用PITR, 这是一个不错的方法。幸运的是, -wal-method=stream实际上在PostgreSQL 10.0或更高版本中已经是默认的了。然而, 在9.6或更早的版本中, 建议使用其前身, 名为-xlogmethod=stream。简而言之: 在PostgreSQL 13.0中不需要再明确地设置这个了。

现在让我们看一下带宽管理。

2.4 减少备份带宽

当pg_basebackup启动时, 它试图尽快完成其工作。如果我们有一个良好的网络连接, pg_basebackup肯定能在一秒内从远程服务器上获取数百兆字节的数据。如果我们的服务器有一个薄弱的I/O系统, 这可能意味着pg_basebackup可以轻易地吸走所有的资源, 而终端用户可能会因为他们的I/O请求太慢而遭遇糟糕的性能

为了控制最大传输速率, pg_basebackup提供了以下内容。

```
-r, --max-rate=RATE
maximum transfer rate to transfer data directory
(in kB/s, or use suffix "k" or "M")
```

当我们创建一个基本的备份时, 我们需要确保主站的磁盘系统能够真正承受住负荷。因此, 调整我们的传输速率可以有很大的意义。

2.5 映射表空间

通常, 如果我们在目标系统上使用相同的文件系统布局, 就可以直接调用pg_basebackup。如果不是这种情况, pg_basebackup允许你将主系统的文件系统布局映射到所需的布局上。-T选项允许我们进行映射。

```
-T, --tablespace-mapping=OLDDIR=NEWDIR
relocate tablespace in OLDDIR to NEWDIR
```

如果你的系统很小, 把所有东西都放在一个表空间里可能是个好主意。

如果I/O不是问题(也许是因为你只管理几千兆字节的数据), 这一点是成立的。

2.6 使用不同的格式

pg_basebackup命令行工具可以创建各种格式。默认情况下, 它将把数据放在一个空目录中。基本上, 它将连接到源服务器并通过网络连接创建一个.tar文件, 然后把数据放到所需的目录中。

这种方法的麻烦在于pg_basebackup会创建很多文件，如果我们想把备份转移到外部备份解决方案（如Tivoli Storage Manager），这并不适合。下面的列表显示了pg_basebackup支持的有效输出格式。

-F, --format=plt output format (plain (default), tar)

要创建一个单一的文件，我们可以使用-F=t选项。默认情况下，它将创建一个名为base.tar的文件，然后可以更容易地管理它。当然，缺点是在执行PITR之前，我们必须再次对文件进行解包。

2.7 测试事务日志归档

在我们深入研究实际的重放过程之前，通过使用简单的ls命令，实际检查归档情况，以确保其工作完美，符合预期，如以下代码所示。

```
[hs@zenbook archive]$ ls -l
total 229384
-rw----- 1 hs staff 16777216 Oct 2 12:38 00000001000000000000000000000007
-rw----- 1 hs staff 339 Oct 2 12:38
00000001000000000000000000000007.00000188.backup
-rw----- 1 hs staff 16777216 Oct 2 12:38 00000001000000000000000000000008
-rw----- 1 hs staff 16777216 Oct 2 12:31 00000001000000000000000000000009
-rw----- 1 hs staff 16777216 Oct 2 12:31 0000000100000000000000000000000A
-rw----- 1 hs staff 16777216 Oct 2 12:31 0000000100000000000000000000000B
-rw----- 1 hs staff 16777216 Oct 2 12:38 0000000100000000000000000000000C
-rw----- 1 hs staff 16777216 Oct 2 12:38 0000000100000000000000000000000D
drwx---- 4 hs staff 136 Oct 2 12:38 archive_status
```

一旦数据库中出现任何严重的活动，WAL文件就应该被送到归档中。

除了仅检查文件之外，以下视图也很有用：

```
test=# \d pg_stat_archiver
 view "pg_catalog.pg_stat_archiver"
 Column | Type | Modifiers
-----+-----+
 archived_count | bigint |
 last_archived_wal | text |
 last_archived_time | timestamp with time zone |
 failed_count | bigint |
 last_failed_wal | text |
 last_failed_time | timestamp with time zone |
 stats_reset | timestamp with time zone |
```

`pg_stat_archiver` 系统视图对于确定归档是否因任何原因停止以及何时停止非常有用。它将告诉我们已经归档的文件的数量 (`archived_count`)。我们还可以看到哪个文件是最后一个，以及事件发生的时间。最后，`pg_stat_archiver` 系统视图可以告诉我们什么时候归档出了问题，这是至关重要的信息。不幸的是，表中没有显示错误代码或信息，但由于 `archive_command` 可以是一个任意的命令，所以很容易记录下

在存档中还有一件事要看。正如我们前面所描述的，检查那些文件是否真的被归档是很重要的。但还有一点。当pg_basebackup命令行工具被调用时，我们会在WAL文件流中看到一个.backup文件。它很小，只包含关于基础备份本身的一些信息--它纯粹是信息性的，重放过程不需要。然而，它给了我们一些重要的线索。当我们以后开始重放事务日志时，我们可以删除所有比.backup文件更早的WAL文件。在这种情况下，我们的备份文件被称为00000000010000000000000007.00000188.backup。这意味着重放过程在文件...0007内的某个地方开始（在...188位置）。这也意味着我们可以删除所有比...0007更早

的文件。旧的WAL文件将不再需要用于恢复。请记住，我们可以保留不止一个备份，所以我只指当前的备份。

现在，归档工作已经完成，我们可以把注意力转向重放过程。

2.8 重放事务日志

让我们总结一下到目前为止的过程。我们调整了postgresql.conf文件（wal_level、max_wal_senders、archive_mode和archive_command），我们在pg_hba.conf文件中允许使用pg_basebackup命令。然后，数据库被重新启动，并成功产生了一个基础备份。

请记住，基础备份只能在数据库完全运行的情况下发生--只需要短暂的重启就可以改变max_wal_sender和wal_level变量。

现在，系统已经正常工作，我们可能会面临崩溃，我们要从中恢复。因此，我们可以执行PITR，尽可能多地恢复数据。我们要做的第一件事是采取基础备份，并把它放在理想的位置。

保存旧的数据库集群可能是一个好主意。即使它已经坏了，我们的PostgreSQL支持公司可能需要它来追踪崩溃的原因。你仍然可以在以后删除它，一旦你让一切重新运行起来。

鉴于前面的文件系统布局，我们可能想做如下事情。

```
cd /some_target_dir  
cp -Rv * /data
```

我们假设新的数据库服务器将位于/data目录下。在你复制基础备份之前，请确保该目录是空的。

在PostgreSQL 13中，有些事情发生了变化：在旧版本中，我们必须配置recovery.conf来控制副本或PITR的一般行为。所有控制这些东西的配置设置都被移到了主配置文件postgresql.conf中。如果你正在运行旧的设置，那么现在是时候改用新的界面了，以确保你的自动化不会被破坏。

那么，让我们看看如何配置重放过程。尝试将restore_command和recovery_target_time放入postgresql.conf中。

```
restore_command = 'cp /archive/%f %p'  
recovery_target_time = '2020-10-02 12:42:00'
```

在修复了postgresql.conf文件后，我们可以简单地启动我们的服务器。输出可能看起来如下。

```
waiting for server to start....  
2020-10-02 12:42:10.085 CEST [52779] LOG: starting PostgreSQL 13.0 on x86_64-apple-darwin17.7.0,  
compiled by Apple LLVM version 10.0.0 (clang-1000.10.44.4), 64-bit  
2020-10-02 12:42:10.092 CEST [52779] LOG: listening on IPv6 address "::1", port 5432  
2020-10-02 12:42:10.092 CEST [52779] LOG: listening on IPv6 address  
"fe80::1%lo0",  
port 5432  
2020-10-02 12:42:10.092 CEST [52779] LOG: listening on IPv4 address "127.0.0.1",  
port 5432  
2020-10-02 12:42:10.093 CEST [52779] LOG: listening on unix socket  
"/tmp/.PGSQL.5432"  
2020-10-02 12:42:10.099 CEST [52780] LOG: database system was interrupted; last  
known up at 2020-10-02 12:38:25 CEST  
cp: /tmp/archive/00000002.history: No such file or directory  
2020-10-02 12:42:10.149 CEST [52780] LOG: entering standby mode  
2020-10-02 12:42:10.608 CEST [52780] LOG: restored log file
```

```
"0000000100000000000000000007" from archive
2020-10-02 12:42:10.736 CEST [52780] LOG: redo starts at 0/7000188
2020-10-02 12:42:10.737 CEST [52780] LOG: consistent recovery state reached at
0/7000260
2020-10-02 12:42:10.737 CEST [52779] LOG: database system is ready to accept
read
only connections
done
server started
2020-10-02 12:42:11.164 CEST [52780] LOG: restored log file
"0000000100000000000000000008" from archive
cp: /tmp/archive/000000010000000000000009: No such file or directory
2020-10-02 12:42:11.292 CEST [52788] LOG: started streaming WAL from primary at
0/9000000 on timeline 1
```

当服务器启动时，有几条信息需要寻找，以确保我们的恢复工作完美。达到一致的恢复状态是最重要的第一条信息。一旦你达到了这一点，你就可以确定你的数据库是一致的，没有被破坏。根据你所选择的时间戳，你可能在最后丢失了一些事务（如果需要的话），但总的来说，你的数据库将是一致的（没有违反键等）。

如果你使用了一个表示未来某个时间点的时间戳，PostgreSQL会抱怨说它找不到下一个WAL文件，并终止重放过程。如果你使用的时间戳是在基础备份结束后到崩溃前的某个地方，你当然不会看到这样的消息。

一旦这个过程完成，服务器将成功启动。

2.9 找到正确的时间戳

到目前为止，我们的进展是假设我们知道我们想要恢复的时间戳，或者我们只是想重放整个事务日志以减少数据损失。然而，如果我们不想重放所有的东西呢？如果我们不知道要恢复到哪个时间点呢？在日常生活中，这其实是一个非常常见的场景。我们的一个开发人员在早上丢失了一些数据，而我们应该让事情恢复正常。问题是这样的：在早上的哪个时间点？一旦恢复结束，就不能轻易重启。一旦恢复完成，系统就会被推广，而一旦被推广，我们就不能继续重播WAL。

然而，我们可以做的是，在没有升级的情况下暂停恢复，检查数据库里面的内容，然后继续。

做到这一点很容易。我们首先要确定的是，在postgresql.conf文件中，hot_standby变量被设置为on。这将确保在数据库仍处于恢复模式时，它是可读的。然后，在postgresql.conf中设置以下变量。

```
recovery_target_action = 'pause'
```

有各种recovery_target_action设置。如果我们使用暂停，PostgreSQL将在所需的时间暂停，让我们检查已经重放的内容。我们可以调整我们想要的时间，重新启动，并再次尝试。另外，我们也可以将该值设置为提升或关闭。

还有一种暂停事务日志回放的方法。基本上，它也可以在执行PITR时使用。但是，在大多数情况下，它是与流式复制一起使用的。下面是在WAL重放期间可以做的事情。

```
postgres=# \x
Expanded display is on.
postgres=# \df *pause*
List of functions
-[ RECORD 1 ]-----+
Schema | pg_catalog
Name   | pg_is_wal_replay_paused
```

```
Result data type | boolean
Argument data types |
Type | normal
-[ RECORD 2 ]-----+
Schema | pg_catalog
Name | pg_wal_replay_pause
Result data type | void
Argument data types |
Type | normal
postgres=# \df *resume*
List of functions
-[ RECORD 1 ]-----+
Schema | pg_catalog
Name | pg_wal_replay_resume
Result data type | void
Argument data types |
Type | normal
```

我们可以调用SELECT pg_wal_replay_pause();命令来停止WAL重放，直到我们调用SELECT pg_wal_replay_resume();命令。

我们的想法是要弄清楚已经重放了多少WAL，并在必要时继续重放。然而，请记住这一点：一旦一个服务器被提升，我们就不能在没有进一步预防措施的情况下继续重放WAL。

正如我们已经看到的，要弄清楚我们需要恢复多远的时间，可能是相当棘手的。因此，PostgreSQL为我们提供了一些帮助。考虑下面这个现实世界的例子：在午夜，我们运行一个夜间进程，在某个通常不为人知的点结束。我们的目标是精确地恢复到夜间进程结束的时间点。问题是这样的。我们如何知道这个过程何时结束？在大多数情况下，这很难搞清楚。那么，为什么不在事务日志中添加一个标记呢？这方面的代码如下。

```
postgres=# SELECT pg_create_restore_point('my_daily_process_ended');
pg_create_restore_point
-----
1F/E574A7B8
(1 row)
```

如果我们的进程一结束就调用这个SQL语句，就可以通过在postgresql.conf文件中添加以下指令，使用事务日志中的这个标签，准确地恢复到这个时间点。

```
recovery_target_name = 'my_daily_process_ended'
```

通过使用这个设置而不是recovery_target_time，重放过程将把我们准确地传送到夜间进程的终点。

当然，我们也可以重放到某个事务ID。然而，在现实生活中，这被证明是困难的，因为管理员很少知道确切的事务ID，因此，这并没有什么实际价值。请记住，设置标记必须在恢复之前完成。这一点很重要。

2.10 清理事务日志存档

到目前为止，数据一直在被写入归档文件，没有注意再次清理归档文件以释放文件系统的空间。PostgreSQL不能为我们做这项工作，因为它不知道我们是否要再次使用存档。因此，我们要负责清理事务日志。当然，我们也可以使用备份工具--然而，重要的是要知道，PostgreSQL没有机会为我们做清理工作。

假设我们想清理一个不再需要的旧交易日志。也许我们想在周围保留几个基础备份，并清理所有不再需要的事务日志，以恢复其中一个备份。

在这种情况下，`pg_archivecleanup`命令行工具正是我们需要的。我们可以简单地将归档目录和备份文件的名称传递给`pg_archivecleanup`命令，它将确保这些文件从磁盘上被删除。使用这个工具使我们的生活变得更容易，因为我们不必自己去计算要保留哪些交易日志文件。下面是它的原理。

```
pg_archivecleanup removes older WAL files from PostgreSQL archives.  
Usage:  
  pg_archivecleanup [OPTION]... ARCHIVELOCATION OLDESTKEPTWALFILE  
Options:  
  -d generate debug output (verbose mode)  
  -n dry run, show the names of the files that would be removed  
  -v, --version output version information, then exit  
  -x EXT clean up files if they have this extension  
  -?, --help show this help, then exit  
For use as archive_cleanup_command in postgresql.conf:  
archive_cleanup_command = 'pg_archivecleanup [OPTION]... ARCHIVELOCATION %r'  
e.g.  
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archiverdir %r'  
or for use as a standalone archive cleaner:  
e.g.  
pg_archivecleanup /mnt/server/archiverdir  
000000010000000000000010.00000020.backup  
Report bugs to <pgsql-bugs@lists.postgresql.org>.  
PostgreSQL home page: <https://www.postgresql.org/>
```

这个工具可以轻松使用，并可在所有平台上使用。

现在我们已经看了一下事务日志存档和PITR，我们可以把注意力集中在当今PostgreSQL世界中最广泛使用的功能之一：流式复制。

3.设置异步复制

流式复制的想法很简单。在最初的基础备份之后，辅助系统可以连接到主系统，实时获取事务日志并加以应用。事务日志重放不再是一个单一的操作，而是一个连续的过程，只要集群存在，就应该一直运行下去。

3.1 执行基本设置

在这一节中，我们将学习如何快速、轻松地设置异步复制。我们的目标是建立一个由两个节点组成的系统。

基本上，大部分的工作已经为WAL归档完成了。然而，为了便于理解，我们将看一下设置流媒体的整个过程，因为我们不能假设WAL运输真的已经按需要设置好了。

首先要做的是进入`postgresql.conf`文件，调整以下参数。

```
wal_level = replica  
max_wal_senders = 10 # or whatever value >= 2  
# this is the default value already in more recent versions  
hot_standby = on # in already a default setting
```

从PostgreSQL 10.0开始，其中一些已经是默认选项。

正如我们之前所做的，wal_level变量必须被调整，以确保PostgreSQL产生足够的事务日志来维持一个从属系统。然后，我们必须配置max_wal_senders变量。当一个从属系统启动和运行时，或者当一个基础备份被创建时，一个WAL发送器进程将与客户端的WAL接收器进程对话。max_wal_senders的设置允许PostgreSQL创建足够的进程来服务这些客户端。

理论上，只需一个WAL发送程序就足够了。然而，这是很不方便的。一个使用wal-method=stream参数的基础备份已经需要两个WAL发送器进程。如果你想同时运行一个slave和执行一个基础备份，已经有三个进程在使用了。因此，请确保你允许PostgreSQL创建足够的进程，以防止无意义的重新启动。

然后，还有hot_standby变量。基本上，主会忽略hot_standby变量，不把它考虑在内。它所做的只是在WAL重放期间使从节点可读。那么，我们为什么要关心呢？请记住，pg_basebackup命令将克隆整个服务器，包括其配置。这意味着，如果我们已经在主服务器上设置了这个值，那么当数据目录被克隆时，从服务器将自动得到它。

设置完postgresql.conf文件后，我们可以把注意力转向pg_hba.conf文件：只要通过添加规则，允许从机执行复制。基本上，这些规则与我们已经看到的PITR的规则是一样的。然后，重新启动数据库服务器，就像对PITR所做的那样

现在，可以在slave上调用pg_basebackup命令了。在我们这样做之前，确保/target目录是空的。如果我们使用的是RPM包，确保你关闭了一个可能正在运行的实例，并清空目录（例如，/var/lib/pgsql/data）。下面的代码显示了如何使用pg_basebackup。

```
pg_basebackup -D /target  
-h master.example.com  
--checkpoint=fast  
--wal-method=stream -R
```

只要把/target目录替换成你想要的目标目录，把master.example.com替换成你的主的IP或DNS名称。--checkpoint=fast参数将触发一个即时检查点。然后，还有--wal-method=stream参数；它将打开两个流。一个将复制数据，而另一个将获取备份运行时创建的WAL数据。

最后，还有-R标志：

```
-R, --write-recovery-conf # write configuration for replication
```

-R标志是一个非常好的功能。pg_basebackup命令可以自动创建从属配置。在旧版本中，它将在recovery.conf文件中添加各种条目。在PostgreSQL 12及以上版本中，它将自动对postgresql.conf进行修改。

```
standby_mode = on primary_conninfo = ' ... '
```

第一个设置说PostgreSQL应该一直重放WAL--如果整个事务日志已经被重放了，它应该等待新的WAL目录到来。第二个设置将告诉PostgreSQL主在哪里。这是一个正常的数据库连接。

从系统也可以连接到其他从系统来流转事务日志。通过简单地从一个从服务器创建基本备份，可以进行级联复制。所以，主服务器在这里真的意味着源服务器。

运行pg_basebackup命令后，可以启动服务。我们应该检查的第一件事是主是否显示了wal sender进程。

```
[hs@zenbook ~]$ ps ax | grep sender
17873 ? Ss 0:00 postgres: wal sender process
ah ::1(57596) streaming 1F/E9000060
```

如果是，slave 也会显示 wal 接收进程：

```
17872 ? Ss 0:00 postgres: wal receiver process
streaming 1F/E9000060
```

如果这些进程在那里，我们就已经在正确的轨道上了，而且复制正在按预期工作。现在双方都在相互交谈，WAL从主流向从。

3.2 提高安全性

到目前为止，我们已经看到，数据是以超级用户的身份流转的。然而，允许超级用户从远程站点访问并不是一个好主意。幸运的是，PostgreSQL允许我们创建一个只允许消费事务日志流的用户，但不能做其他事情。

创建一个仅用于流式传输的用户很容易。下面是它的工作原理：

```
test=# CREATE USER repl LOGIN REPLICATION;
CREATE ROLE
```

通过将复制分配给用户，有可能只用于流式传输--其他一切都被禁止了。

强烈建议不要使用你的超级用户账户来设置流式传输。只需将配置文件改为新创建的用户。不暴露超级用户账户将极大地提高安全性，就像给复制用户一个密码一样。

3.3 停止和恢复复制

一旦设置了流式复制，它就能完美地工作，不需要管理员过多的干预。然而，在某些情况下，停止复制并在以后恢复复制可能是有意义的。为什么有人想这样做呢？

考虑以下用例：你负责一个主/从设置，它正在运行一个不合格的内容管理系统（CMS）或一些可疑的论坛软件。假设你想把你的应用程序从糟糕的CMS 1.0更新到糟糕的CMS 2.0。一些变化将在你的数据库中执行，这些变化将立即被复制到从属数据库中。如果升级过程中出现了错误怎么办？由于流式传输，错误将立即复制到两个节点

为了避免即时复制，我们可以停止复制，然后根据需要恢复。在我们的CMS更新案例中，我们可以简单地做以下事情。

1. 停止复制。
2. 在主服务器上执行应用程序更新。
3. 检查我们的应用程序是否仍然工作。如果是，恢复复制。如果不是，则故障转移到副本，该副本仍有旧的数据。

通过这种机制，我们可以保护我们的数据，因为我们可以回退到问题发生前的数据。在本章的后面，我们将学习如何将一个从属服务器提升为新的主服务器。

现在的主要问题是：我们如何才能停止复制？下面是它的工作方式。在备用机上执行以下一行。

```
test=# SELECT pg_wal_replay_pause();
```

这一行将停止复制。请注意，事务日志仍然会从主服务器流向从节点--只是重放过程被停止了。你的数据仍然受到保护，因为它被持久化在从服务器上。在服务器崩溃的情况下，没有数据会丢失。

请记住，重放过程必须在从机上停止。否则，PostgreSQL将抛出一个错误。

```
ERROR: recovery is not in progress
HINT: Recovery control functions can only be executed during recovery.
```

一旦要恢复复制，在从属机构上需要有以下一行。

```
SELECT pg_wal_replay_resume();
```

PostgreSQL 将再次开始重放 WAL。

3.4 检查复制以确保可用性

每个管理员的核心工作之一是确保复制在任何时候都能保持正常运行。如果复制出现故障，如果主服务器崩溃，数据就有可能丢失。因此，保持对复制的关注是绝对必要的。

幸运的是，PostgreSQL提供了系统视图，使我们能够深入了解正在发生的事情。其中一个视图是 pg_stat_replication。

```
test=# \d pg_stat_replication
          View "pg_catalog.pg_stat_replication"
 Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+-----+
 pid   | integer | | |
 usesysid | oid | | |
 username | name | | |
 application_name | text | | |
 client_addr | inet | | |
 client_hostname | text | | |
 client_port | integer | | |
 backend_start | timestamp with time zone | | |
 backend_xmin | xid | | |
 state | text | | |
 sent_lsn | pg_lsn | | |
 write_lsn | pg_lsn | | |
 flush_lsn | pg_lsn | | |
 replay_lsn | pg_lsn | | |
 write_lag | interval | | |
 flush_lag | interval | | |
 replay_lag | interval | | |
 sync_priority | integer | | |
 sync_state | text | | |
 reply_time | timestamp with time zone | | |
```

pg_stat_replication视图将包含关于发送者的信息。我不想在这里使用主站这个词，因为从站可以连接到其他一些从站。有可能建立一个服务器树。在服务器树的情况下，主站将只拥有它直接连接的从站的信息。

在这个视图中，我们将看到的第一件事是WAL发送进程的进程ID。它可以帮助我们在出错的情况下识别该进程。通常不会出现这种情况。然后，我们将看到从属进程用来连接到其发送服务器的用户名。client_*字段将表明从属进程的位置。我们将能够从这些字段中提取网络信息。backend_start字段显示了从属设备何时开始从我们的服务器进行流式传输。

然后，有一个神奇的backend_xmin字段。假设你正在运行一个主/从设置。可以告诉从属系统向主系统报告其事务ID，这背后的想法是延迟 master 上的清理，这样数据就不会从 slave 上运行的事务中获取state字段通知我们关于服务器的状态。如果我们的系统没有问题，该字段将包含流。否则，需要仔细检查。

接下来的四个字段是真正重要的。sent_lsn字段，以前是send_location字段，表示有多少WAL已经到达对方，这意味着这些字段已经被WAL接收方接受。我们可以用它来计算出有多少数据已经到达了从机。然后，是write_lsn字段，以前是write_location字段。一旦WAL被接受，它就被传递给操作系统。Write_lsn字段将告诉我们，WAL的位置已经安全地传到了操作系统。flush_lsn字段，也就是以前的flush_location字段，将知道数据库有多少WAL已经被刷到磁盘上了。

最后是replay_lsn，以前是replay_location字段。WAL已经到了备用机的磁盘上，但这并不意味着PostgreSQL已经重放或已经被最终用户看到。假设复制被暂停了。数据仍然会流向备用机。然而，它将在以后被应用。replay_lsn字段将告诉我们有多少数据是已经可见的。

在PostgreSQL 10.0中，更多的字段被添加到pg_stat_replication中；*_lag字段表示从属系统的延迟，并提供了一个方便的方法来查看从属系统的落后程度。

这些字段的间隔不同，这样我们可以更清楚地看到时间上的差异。

最后，PostgreSQL告诉我们复制是同步的还是异步的。

如果我们还在PostgreSQL 9.6上，我们可能会发现计算发送和接收服务器之间的字节差是很有用的。

9.6版的*_lag字段还不能做到这一点，所以有字节的差异会非常有利。下面是它的工作原理。

```
SELECT client_addr, pg_current_wal_location() - sent_location AS diff
FROM pg_stat_replication;
```

当在主服务器上运行时，pg_current_wal_location()函数返回当前的交易日志位置。PostgreSQL 9.6有一个特殊的数据类型用于事务日志位置，叫做pg_lsn。它有几个运算符，这里用将主服务器的WAL位置中减去从属服务器的WAL位置。因此，这里概述的视图返回两个服务器之间的差异，单位是字节（复制延迟）。

注意，这个语句只在PostgreSQL 10中起作用。在较早的版本中，这个函数曾经被称为pg_current_xlog_location()。

pg_stat_replication系统视图包含发送方的信息，而pg_stat_wal_receiver系统视图将为我们提供接收方的类似信息。

```
test=# \d pg_stat_wal_receiver
  View "pg_catalog.pg_stat_wal_receiver"
 Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
 pid   | integer |  || |
 status | text |  || |
 receive_start_lsn | pg_lsn |  || |
 receive_start_tli | integer |  || |
 written_lsn | pg_lsn |  || |
 flushed_lsn | pg_lsn |  || |
```

```
received_tli | integer | |
last_msg_send_time | timestamp with time zone | |
last_msg_receipt_time | timestamp with time zone | |
latest_end_lsn | pg_lsn | |
latest_end_time | timestamp with time zone | |
slot_name | text | |
sender_host | text | |
sender_port | integer | |
conninfo | text | |
```

在WAL接收器进程的进程ID之后，PostgreSQL会向你提供该进程的状态。然后，receive_start_lsn字段将告诉你WAL接收器开始的事务日志位置，而receive_start_tli字段将告知你WAL接收器启动时使用的时间轴。

written_lsn和flushed_lsn字段包含了关于WAL位置的信息，分别是已经写入和刷入磁盘的。然后我们得到了一些关于时间的信息，以及插槽和连接的信息。

一般来说，许多人发现阅读pg_stat_replication系统视图比阅读pg_stat_wal_receiver视图更容易，而且大多数工具都是围绕pg_stat_replication视图建立的。

3.5 执行故障转移和了解时间表

一旦创建了主/从设置，它通常会在很长一段时间内完美无缺地工作。然而，一切都可能失败，因此，了解如何用一个备份系统取代一个失败的服务器是很重要的。

PostgreSQL使故障转移和推广变得很容易。基本上，我们所要做的就是使用pg_ctl参数来告诉一个副本提升自己。

```
pg_ctl -D data_dir promote
```

服务器将断开自己与主站的连接，并立即执行升级。请记住，从服务器在升级时可能已经支持数千个只读连接。PostgreSQL的一个很好的特点是，在晋升过程中，所有打开的连接都会变成读/写连接--甚至不需要重新连接。

请注意，PostgreSQL 12也能够使用普通的SQL将数据库从从站提升到主站。只要使用SELECT pg_promote();。

当提升一个服务器时，PostgreSQL将递增时间线：如果你建立了一个全新的服务器，它将在时间线1。如果从该服务器克隆出一个从属服务器，它将和它的主服务器在同一时间线。所以，两个盒子都将在时间线1中。如果从机被提升为独立的主机，它将进入时间线2。

时间线对于PITR来说特别重要。假设我们在午夜时分创建一个基础备份。在上午12点，从属系统被提升。在下午3点，有东西崩溃了，我们想恢复到下午2点。我们将重放基础备份后创建的事务日志，并跟踪我们所需服务器的WAL流，因为这两个节点在上午12点开始出现了分歧。

时间线的变化也将在事务日志文件的名称中可见。下面是一个时间线1的WAL文件的例子。

```
000000010000000000000000F5
```

如果时间线切换到2，新文件名将如下所示：

```
000000020000000000000000F5
```

正如你所看到的，来自不同时间线的WAL文件理论上可以存在于同一个存档目录中。

3.6 管理冲突

到目前为止，我们已经学到了很多关于复制的知识。然而，看一看复制冲突是很重要的。出现的主要问题是：冲突首先是如何发生的？

考虑以下示例：

Master	Slave
DROP TABLE tab;	BEGIN;
	SELECT ... FROM tab WHERE ...
	... running ...
	... conflict happens ...
	... transaction is allowed to continue for 30 seconds ...
	... conflict is resolved or ends before timeout ...

这里的问题是，主站不知道从站上有一个事务发生。因此，DROP TABLE命令不会阻塞，直到读取事务消失。如果这两个事务发生在同一个节点上，当然会是这样的情况。然而，我们在这里看到的是两个服务器。DROP表命令将正常执行，而杀死磁盘上那些数据文件的请求将通过事务日志到达从机。从机没有麻烦：如果表从磁盘上被删除，SELECT子句就必须死亡--如果从机在应用WAL之前等待SELECT子句的完成，它可能会无可奈何地落后。

理想的解决方案是可以使用配置变量来控制的折中方案。

```
max_standby_streaming_delay = 30s
# max delay before canceling queries
# when reading streaming WAL;
```

我们的想法是，在解决冲突之前，先等待30秒，在从机上杀死查询。根据我们的应用，我们可能想把这个变量改成一个更积极的设置。注意，30秒是针对整个复制流，而不是针对单个查询。可能因为其他查询已经等待了一段时间，所以单个查询被提前杀死。

虽然DROP TABLE命令显然是一种冲突，但有一些操作却不那么明显。下面是一个例子。

```
BEGIN;
...
DELETE FROM tab WHERE id < 10000;
COMMIT;
...
VACUUM tab;
```

再一次，让我们假设在slave上有一个长期运行的SELECT子句。在这里，DELETE子句显然不是问题，因为它只是将该行标记为已删除--它实际上并没有删除它。提交也不是一个问题，因为它只是将事务标记为完成。从物理上讲，该行仍然在那里。

当VACUUM等操作启动时，问题就开始了。它将破坏磁盘上的行。当然，这些变化会进入WAL，并最终到达从机，这时从机就会出现问题。

为了防止由标准OLTP工作负载引起的典型问题，PostgreSQL开发团队引入了一个配置变量。

```
hot_standby_feedback = off
# send info from standby to prevent
# query conflicts
```

如果这个设置是打开的，从属系统将定期向主系统发送最古老的事务ID。然后，VACUUM将知道在系统的某个地方有一个较长的事务正在进行，并将清理年龄推迟到以后安全的时候再清理这些行。事实上，hot_standby_feedback参数引起的效果与主站的长事务相同。

我们可以看到，默认情况下，hot_standby_feedback参数是关闭的。为什么会这样呢？嗯，有一个很好的理由：如果它是关闭的，一个从站不会对主站产生实际影响。事务日志流不会消耗大量的CPU功率，使流式复制变得廉价和高效。然而，如果一个从站（甚至可能不在我们的控制之下）保持事务开放时间过长，我们的主站可能会因为清理过晚而受到表膨胀的影响。在默认设置中，这比减少冲突更不可取。

如果hot_standby_feedback = on，通常可以避免99%的OLTP相关冲突，如果你的交易时间超过几毫秒，这一点就特别重要。

3.7 使复制更可靠

在本章中，我们已经看到，设置复制是很容易的，不需要花费很多精力。然而，总有一些角落的情况会导致操作上的挑战。其中一个角落案例就是关于事务日志的保留。

考虑以下场景：

- 获取了一个基本的备份。
- 备份后，1小时内没有任何反应
- 从机启动。

请记住，主站并不太关心从站的存在。因此，从机启动所需的事务日志可能已经不存在于主机上了，因为它可能已经被检查点移除。问题是，需要重新同步才能启动从属系统。在一个多TB的数据库中，这显然还是一个问题。

这个问题的一个潜在解决方案是使用wal_keep_segments设置。

```
wal_keep_segments = 0 # in logfile segments, 16MB each; 0 disables
```

默认情况下，PostgreSQL保留足够的事务日志，以便在意外的崩溃中存活，但不会多到哪里去。通过wal_keep_segments设置，我们可以告诉服务器保留更多的数据，这样即使从属系统落后，也能赶上。

重要的是要记住，服务器落后不仅是因为它们太慢或太忙--在许多情况下，延迟发生是因为网络太慢。假设你正在为一个1TB的表创建一个索引。PostgreSQL会对数据进行排序，当索引真正建立时，也会被发送到交易日志中。试想一下，当几百兆的WAL通过一条也许只能处理1GB左右的线路发送时会发生什么。许多千兆字节的数据的丢失可能就是这个后果，而且会在几秒钟内发生。因此，调整wal_keep_segments设置不应该关注典型的延迟，而应该关注管理员可以容忍的最高延迟（也许有一定的安全系数）。

为wal_keep_segments设置投资一个合理的高设置是非常有意义的，我建议确保周围总是有足够的数据。

解决事务日志耗尽问题的另一个办法是复制槽，这将在本章后面介绍。

4.升级到同步复制

到目前为止，我们已经对异步复制进行了合理的阐述。然而，异步复制意味着允许从机上的提交在主机上的提交之后发生。如果主站崩溃了，即使复制正在发生，还没有到从站的数据也可能会丢失。

同步复制就是为了解决这个问题--如果PostgreSQL同步复制，一个提交必须由至少一个副本写入到磁盘上才能在主服务器上通过。因此，同步复制基本上可以大大降低数据丢失的几率。

在PostgreSQL中，配置同步复制很容易。只有两件事要做（按任何顺序）。

- 在主站的postgresql.conf文件中调整synchronous_standby_names设置。
- 在副本的配置文件中的primary_conninfo参数中增加一个application_name设置。

让我们从 master 上的 postgresql.conf 文件开始：

```
synchronous_standby_names = ''  
# standby servers that provide sync rep  
# number of sync standbys and comma-separated  
# list of application_name  
# from standby(s); '*' = all
```

如果我们输入'*'，所有的节点都将被视为同步候选。然而，在现实生活中，更有可能的是，只有几个节点会被列出。下面是一个例子。

```
synchronous_standby_names = 'slave1, slave2, slave3'
```

在前面的案例中，我们得到了三个同步的候选人。现在，我们必须改变配置文件，加入application_name。

```
primary_conninfo = '... application_name=slave2'
```

复制体现在将作为slave2连接到master。master将检查它的配置，发现slave2是列表中第一个构成可行的slave。因此，PostgreSQL将确保只有在从属系统确认事务存在的情况下，master的提交才会成功。

现在，让我们假设slave2由于某种原因发生故障。PostgreSQL将尝试把另外两个节点中的一个变成同步的备用节点。问题是这样的：如果没有其他服务器怎么办？

在这种情况下，如果一个事务应该是同步的，PostgreSQL将永远等待提交。除非至少有两个可行的节点可用，否则PostgreSQL不会继续提交。记住，我们已经要求PostgreSQL在至少两个节点上存储数据--如果我们不能在任何时间点提供足够的主机，那就是我们的错。在现实中，这意味着同步复制最好在至少三个节点上实现--一个主节点和两个从节点--因为总是有可能有一个主机被丢失。

谈到主机故障，在这一点上有一件重要的事情需要注意--如果一个同步伙伴在提交过程中死亡，PostgreSQL将等待它回来。另外，同步提交也可能发生在其他潜在的同步伙伴身上。最终用户可能根本不会注意到同步伙伴已经改变。

在某些情况下，仅仅将数据存储在两个节点上可能是不够的：也许我们想进一步提高安全性，将数据存储在更多的节点上。为了达到这个目的，我们可以在PostgreSQL 9.6或更高版本中利用以下语法。

```
synchronous_standby_names =  
'4(slave1, slave2, slave3, slave4, slave5, slave6)'
```

当然，这也是有代价的--请记住，如果我们增加越来越多的同步复制，速度就会下降。世界上没有免费的午餐。PostgreSQL提供了几种方法来控制性能开销，我们将在下一节讨论。

在PostgreSQL 10.0中，甚至增加了更多的功能。让我们来看看协议的相关部分。

```
[FIRST] num_sync ( standby_name [, ...] )
ANY num_sync ( standby_name [, ...] )
standby_name [, ...]
```

ANY和FIRST关键字已经被引入。FIRST允许你设置服务器的优先级，而ANY让PostgreSQL在提交同步事务时有更多的灵活性。

4.1 调整耐久度

在本章中，我们已经看到，数据要么是同步复制，要么是异步复制。然而，这并不是一个全局性的东西。为了确保良好的性能，PostgreSQL允许我们以一种非常灵活的方式来配置事物。我们有可能同步或异步地复制所有的东西，但是在很多情况下，我们可能想以更精细的方式来做的事情。这正是需要synchronous_commit设置的时候。

假设在postgresql.conf文件中已经配置了同步复制、application_name设置和synchronous_standby_names设置，synchronous_commit设置将提供以下选项

- off：这基本上是一个异步复制。WAL不会立即被刷新到主站的磁盘上，主站不会等待从站将所有内容写入磁盘。如果主站发生故障，一些数据可能会丢失（最多三次 - wal_writer_delay）。
- local：事务日志在主站提交时被刷新到磁盘。然而，主站并不等待从站（异步复制）。
- remote_write：remote_write 的设置已经使 PostgreSQL 同步复制。然而，只有主站会将数据保存到磁盘。对于从站来说，把数据发送到操作系统就足够了。我们的想法是不要等待第二次磁盘刷新来加速事情。两个存储系统完全在同一时间崩溃的可能性非常小。因此，数据丢失的风险接近于零。
- on：在这种情况下，如果主站和从站都成功地将事务刷到了磁盘上，那么事务就是OK的。除非数据被安全地存储在两个服务器上（或更多，取决于配置），否则应用程序不会收到提交。
- remote_apply：虽然在上确保了数据被安全地存储在两个节点上，但它并不能保证我们可以马上简单地进行负载平衡。数据被刷新在磁盘上的事实并不能保证用户已经可以看到数据。例如，如果有冲突，从机将停止事务重放--然而，在冲突期间，事务日志仍然被发送到从机，并被刷新到磁盘。简而言之，数据可能会被刷新在从机上，即使它对终端用户还不可见。

remote_apply选项解决了这个问题。它确保数据在副本上必须是可见的，这样下一个读取请求就可以在从属设备上安全地执行，从属设备已经可以看到对主设备所做的修改并将其暴露给最终用户。当然，remote_apply选项是复制数据的最慢方式，因为它要求从属设备已经将数据暴露给终端用户。

在PostgreSQL中，synchronous_commit参数不是一个全局值。它可以在不同层次上进行调整，就像许多其他设置一样。我们可能想做如下的事情。

```
test=# ALTER DATABASE test SET synchronous_commit TO off;
ALTER DATABASE
```

有时，只有一个数据库应该以某种方式进行复制。如果我们作为一个特定的用户连接，也可以只进行同步复制。最后，但同样重要的是，也可以告诉单个事务如何提交。通过实时调整synchronous_commit参数，我们甚至可以在每个事务层面上控制事情。

例如，考虑以下两种情况：

- 写入一个日志表，我们可能想使用异步提交，因为我们想快速完成。
- 存储信用卡付款，我们想保证安全，所以可能需要同步交易。

我们可以看到，非常相同的数据库可能有不同的要求，这取决于哪些数据被修改。因此，在事务层面改变数据是非常有用的，有助于提高速度。

5.利用复制槽

在介绍了同步复制和动态可调的持久性之后，我想重点介绍一个叫做复制槽的功能

复制槽的目的是什么？让我们考虑下面的例子。有一个主站和一个从站。在主服务器上，一个大型交易被执行，网络连接的速度不足以及时运送所有的数据。在某个时候，主站删除了它的事务日志（检查点）。如果从属系统落后太多，就需要重新同步。正如我们已经看到的，wal_keep_segments设置可以用来减少复制失败的风险。问题是：wal_keep_segments设置的最佳值是什么？当然，越多越好，但多少才是最好的？

复制槽将为我们解决这个问题：如果我们使用复制槽，主站只能在事务日志被所有的复制体消耗完后再回收它。这里的好处是，从机永远不会落后到需要重新同步的程度。

问题是，如果我们在没有告诉主服务器的情况下关闭了一个副本怎么办？主服务器会永远保留交易日志，主服务器上的磁盘最终会被填满，造成不必要的停机时间。

为了减少主站的这种风险，复制槽应该只与适当的监控和警报结合使用。只是有必要留意那些有可能导致问题或可能不再使用的开放复制槽。

在 PostgreSQL 中，有两种类型的复制槽：

- 物理复制槽
- 逻辑复制槽

物理复制槽可用于标准流复制。它们将确保数据不被过早回收。逻辑复制槽做同样的事情。然而，它们被用于逻辑解码。逻辑解码背后的想法是让用户有机会附加到事务日志上，并用插件对其进行解码。因此，逻辑事务槽是数据库实例的某种-f tail。它允许用户提取对数据库所做的改变--因此也是对事务日志的改变--以任何格式和任何目的。在许多情况下，逻辑复制槽被用于逻辑复制

5.1 处理物理复制槽

为了利用复制槽，必须对postgresql.conf文件进行修改，具体如下。

```
wal_level = logical  
max_replication_slots = 5 # or whatever number is needed
```

对于物理槽，逻辑槽是没有必要的 - 一个副本就足够了。然而，对于逻辑槽，我们需要一个更高的wal_level设置。那么，如果我们使用的是PostgreSQL 9.6或以下版本，就必须改变max_replication_slots的设置。PostgreSQL 10.0已经有一个改进的默认设置。基本上，我们只是输入一个符合我们目的的数字。我的建议是增加一些备用槽，这样我们就可以很容易地附加更多的消费者，而不必沿途重启服务器

重新启动后，可以创建插槽：

```
test=# \x  
Expanded display is on.  
test=# \df *create*physicalslot  
List of functions  
-[ RECORD 1 ]-----  
--  
-----  
Schema | pg_catalog  
Name  | pg_create_physical_replication_slot  
Result data type | record  
Argument data types | slot_name name, immediately_reserve boolean DEFAULT false,  
temporary boolean DEFAULT false, OUT slot_name name, OUT lsn
```

```
pg_1sn
Type | func
```

pg_create_physical_replication_slot函数是用来帮助我们创建槽的。它可以用两个参数之一来调用：如果只传递一个槽的名字，那么这个槽将在第一次使用时被激活。如果第二个参数是true，槽将立即开始保存交易日志。

```
test=# SELECT * FROM pg_create_physical_replication_slot('some_slot_name',
true);
slot_name | lsn
-----+-----
some_slot_name | 0/EF8AD1D8
(1 row)
```

要想知道哪些插槽在主站上处于活动状态，可以考虑运行以下SQL语句。

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pg_replication_slots;
-[ RECORD 1 ]-----+
slot_name | some_slot_name
plugin |
slot_type | physical
datoid |
database |
temporary | f
...
```

视图告诉我们很多关于槽的信息。它包含关于正在使用的槽的类型、事务日志位置等信息。

为了利用这个槽，我们所要做的就是把它添加到配置文件中，如下所示。

```
primary_slot_name = 'some_slot_name'
```

一旦流媒体被重新启动，该插槽将被直接使用，并将保护复制。如果我们不想要我们的槽位了，我们可以轻松地放弃它。

```
test=# \df *drop*slot*
List of functions
-[ RECORD 1 ]-----+
Schema | pg_catalog
Name | pg_drop_replication_slot
Result data type | void
Argument data types | name
Type | normal
```

当一个槽被放弃时，不再有逻辑槽和物理槽的区别。只需将槽的名称传递给函数并执行它。

当槽被放弃时，不允许任何人使用该槽。否则，PostgreSQL会有充分的理由出错。

5.2 处理逻辑复制槽

逻辑复制槽对逻辑复制至关重要。由于本章篇幅有限，很遗憾，我们不可能涵盖逻辑复制的所有方面。然而，我想概述一些基本概念，这些概念对逻辑解码至关重要，因此也对逻辑复制至关重要。

如果我们想创建一个复制槽，这里是如何进行的。这里需要的函数需要两个参数：第一个参数将定义复制槽的名称，而第二个参数将携带用于解码交易日志的插件。它将决定PostgreSQL将使用何种格式来返回数据。让我们来看看如何制作一个复制槽。

```
test=# SELECT *
  FROM pg_create_logical_replication_slot('logical_slot', 'test_decoding');
slot_name | lsn
-----+-----
logical_slot | 0/EF8AD4B0
(1 row)
```

我们可以使用先前使用的相同命令来检查槽的存在。为了检查槽的真正作用，可以创建一个小测试。

```
test=# CREATE TABLE t_demo (id int, name text, payload text);
CREATE TABLE
test=# BEGIN;
BEGIN
test=# INSERT INTO t_demo
VALUES (1, 'hans', 'some data');
INSERT 0 1
test=# INSERT INTO t_demo VALUES (2, 'paul', 'some more data');
INSERT 0 1
test=# COMMIT;
COMMIT
test=# INSERT INTO t_demo VALUES (3, 'joe', 'less data');
INSERT 0 1
```

请注意，有两个事务被执行。现在可以从槽中提取对这些事务的修改。

```
test=# SELECT pg_logical_slot_get_changes('logical_slot', NULL, NULL);
pg_logical_slot_get_changes
-----
(0/EF8AF5B0,606546,"BEGIN 606546")
(0/EF8CCCA0,606546,"COMMIT 606546")
(0/EF8CCCD8,606547,"BEGIN 606547")
(0/EF8CCCD8,606547,"table public.t_demo: INSERT: id[integer]:1
name[text]:'hans' payload[text]:'some data'")
(0/EF8CCD60,606547,"table public.t_demo: INSERT: id[integer]:2
name[text]:'paul' payload[text]:'some more data'")
(0/EF8CCDE0,606547,"COMMIT 606547")
(0/EF8CCE18,606548,"BEGIN 606548")
(0/EF8CCE18,606548,"table public.t_demo: INSERT: id[integer]:3
name[text]:'joe' payload[text]:'less data'")
(0/EF8CCE98,606548,"COMMIT 606548")
(9 rows)
```

这里使用的格式取决于我们之前选择的输出插件。有各种用于PostgreSQL的输出插件，例如wal2json

如果使用默认值，逻辑流将包含真实值，而不仅仅是函数。逻辑流有最终进入底层表的数据。

另外，请记住，一旦槽被消耗，就不再返回数据。

```
test=# SELECT pg_logical_slot_get_changes('logical_slot', NULL, NULL);
pg_logical_slot_get_changes
-----
(0 rows)
```

因此，第二次调用的结果集是空的。如果我们想重复地获取数据，PostgreSQL提供了pg_logical_slot_peek_changes函数。它的工作原理与pg_logical_slot_get_changes函数一样，但确保数据在槽中仍然可用。

当然，使用普通SQL并不是消费事务日志的唯一方法。还有一个叫做pg_recvlogical的命令行工具。它可以与在整个数据库实例上使用-f tail相比，并实时接收数据流

让我们使用以下命令启动 pg_recvlogical 工具：

```
[hs@zenbook ~]$ pg_recvlogical -s logical_slot -P test_decoding -d test -U
postgres --start -f -
```

在这种情况下，该工具连接到测试数据库并从logical_slot消耗数据。-f意味着数据流将被发送到stdout。让我们杀死一些数据。

```
test=# DELETE FROM t_demo WHERE id < random()*10;
DELETE 3
```

这些变化将进入事务日志。然而，在默认情况下，数据库只关心删除后的表是什么样子的。它知道哪些块必须被触及等等，但它不知道它以前是什么样子。

```
BEGIN 606549
table public.t_demo: DELETE: (no-tuple-data)
table public.t_demo: DELETE: (no-tuple-data)
table public.t_demo: DELETE: (no-tuple-data)
COMMIT 606549
```

因此，输出结果是相当无意义的。为了解决这个问题，下面这一行就来了。

```
test=# ALTER TABLE t_demo REPLICA IDENTITY FULL;
ALTER TABLE
```

如果表被重新填充数据并再次被删除，事务日志流将看起来如下。

```
BEGIN 606558
table public.t_demo: DELETE: id[integer]:1 name[text]:'hans'
payload[text]:'some data'
table public.t_demo: DELETE: id[integer]:2 name[text]:'paul'
payload[text]:'some more data'
table public.t_demo: DELETE: id[integer]:3 name[text]:'joe'
payload[text]:'less data'
COMMIT 606558
```

现在，所有的变化都在。接下来让我们看看逻辑复制槽的情况。

5.3 逻辑复制槽的用例

逻辑复制槽有多种用例。最简单的用例是如下所示。可以从服务器上获取所需格式的数据，并用于审计、调试，或简单地监控数据库实例。

下一个合乎逻辑的步骤是把这个变化流用于复制。双向复制（BDR）等解决方案完全基于逻辑解码，因为二进制级别的变化在多主复制中不起作用。

最后，有时需要在不停机的情况下进行升级。记住，二进制事务日志流不能用于在不同版本的PostgreSQL之间进行复制。因此，未来的PostgreSQL版本将支持一个叫做pglogical的工具，它有助于在不停机的情况下进行升级。

在本节中，你学习了逻辑解码和其他一些基本技术。现在让我们看一下CREATE PUBLICATION和CREATE SUBSCRIPTION命令。

6.利用CREATE PUBLICATION和CREATE SUBSCRIPTION命令

对于10.0版本，PostgreSQL社区创建了两个新命令。CREATE PUBLICATION和CREATE SUBSCRIPTION。这些可以用于逻辑复制，这意味着你现在可以有选择地复制数据，实现接近零停机时间的升级。到目前为止，二进制复制和事务日志复制已经完全涵盖。然而，有时候，我们可能不想复制整个数据库实例--复制一两个表可能就够了。这正是逻辑复制的正确使用时机。

在开始之前，首先要做的是在postgresql.conf中把wal_level改为logical，如下所示，然后重新启动。

```
wal_level = logical
```

然后，我们可以创建一个简单的表：

```
test=# CREATE TABLE t_test (a int, b int);
CREATE TABLE
```

同样的表布局也必须存在于第二个数据库中，以使其发挥作用。PostgreSQL不会自动为我们创建这些表。

```
test=# CREATE DATABASE repl;
CREATE DATABASE
```

创建数据库后，可以添加相同的表：

```
repl=# CREATE TABLE t_test (a int, b int);
CREATE TABLE
```

这里的目标是将测试数据库中的t_test表的内容发布到其他地方。在这种情况下，它将被简单地复制到同一实例的数据库中。为了发布这些变化，PostgreSQL提供了CREATE PUBLICATION命令。

```
test=# \h CREATE PUBLICATION
Command: CREATE PUBLICATION
Description: define a new publication
Syntax:
CREATE PUBLICATION name
[ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
| FOR ALL TABLES ]
[ WITH ( publication_parameter [= value] [, ...] ) ]
URL: https://www.postgresql.org/docs/13/sql-createpublication.html
```

语法其实很简单。我们所需要的只是一个名称和一个系统应该复制的所有表的列表。

```
test=# CREATE PUBLICATION pub1 FOR TABLE t_test;
CREATE PUBLICATION
```

现在，可以创建订阅。同样，语法非常简单明了：

```
test=# \h CREATE SUBSCRIPTION
Command: CREATE SUBSCRIPTION
Description: define a new subscription
Syntax:
CREATE SUBSCRIPTION subscription_name
CONNECTION 'conninfo'
PUBLICATION publication_name [, ...]
[ WITH ( subscription_parameter [= value] [, ...] ) ]
URL: https://www.postgresql.org/docs/13/sql-createsubscription.html
```

基本上，直接创建一个订阅是绝对没有问题的。但是，如果我们在同一个实例里面从测试数据库到复制数据库玩这个游戏，就必须手动创建使用中的复制槽。否则，CREATE SUBSCRIPTION将永远不会完成。

```
test=# SELECT pg_create_logical_replication_slot('sub1', 'pgoutput');
pg_create_logical_replication_slot
-----
(sub1,0/27E2B2D0)
(1 row)
```

在这种情况下，在主数据库上创建的槽的名称叫做sub1。然后，我们需要连接到目标数据库并运行以下命令。

```
rep1=# CREATE SUBSCRIPTION sub1
CONNECTION 'host=localhost dbname=test user=postgres'
PUBLICATION pub1
WITH (create_slot = false);
CREATE SUBSCRIPTION
```

当然，我们必须调整我们数据库的CONNECTION参数。然后，PostgreSQL将同步数据，我们就完成了。

注意，create_slot = false的使用只是因为测试是在同一个数据库服务器实例内运行。如果我们碰巧使用不同的数据库，就不需要手动创建槽，也不需要create_slot = false。

7.总结

在这一章中，我们了解了PostgreSQL复制的最重要特性，比如流式复制和复制冲突。然后我们了解了PITR，以及复制槽。请注意，一本关于复制的书除非跨度在400页左右，否则永远不会完整，但我们已经学到了每个管理员应该知道的最重要的东西。不过，你已经学会了如何设置复制，了解了最重要的方面。

下一章，第11章，决定有用的扩展，是关于PostgreSQL的有用扩展。我们将了解那些提供更多功能并被业界广泛采用的扩展。

8.问题

- 逻辑复制的目的是什么？
- 同步复制的性能影响是什么？
- 为什么不总是使用同步复制？

这些问题的答案可以在 GitHub 仓库中找到 (<https://github.com/PacktPublishing/Mastering-PostgreSQL-13-Fourth-Edition>)

决定有用的扩展

1. 了解扩展是如何工作的

1.1 检查可用的扩展

2. 利用contrib模块

2.1 使用 adminpack 模块

2.2 应用布隆过滤器

2.3 部署 btree_gist 和 btree_gin

2.4 dblink——考虑逐步淘汰

2.5 使用 file_fdw 获取文件

2.6 使用 pageinspect 检查存储

2.7 使用 pg_buffercache 研究缓存

2.8 使用 pgcrypto 加密数据

2.9 使用 pg_prewarm 预热缓存

2.10 使用 pg_stat_statements 检查性能

2.11 使用 pgstattuple 检查存储

2.12 使用 pg_trgm 进行模糊搜索

2.13 使用 postgres_fdw 连接到远程服务器

2.13.1 处理错误和拼写错误

3. 其他有用的扩展

4. 总结

在第10章 "理解备份和复制" 中，我们的重点是复制、交易日志运输和逻辑解码。在看了大部分与管理有关的主题后，现在的目标是瞄准一个更广泛的主题。在PostgreSQL的世界里，许多事情是通过扩展来完成的。扩展的好处是，可以在不影响PostgreSQL核心的情况下增加功能。用户可以从相互竞争的扩展中进行选择，找到最适合他们的东西。其理念是保持核心的纤细，相对容易维护，并为未来做好准备。

在这一章中，将讨论一些最常用的PostgreSQL的扩展。然而，在深入探讨这个问题之前，我想说明的是，本章只介绍了我个人认为有用的扩展列表。现在有这么多的模块，不可能以合理的方式将它们全部涵盖。每天都有信息发布，有时对于一个专业人士来说，甚至很难了解所有的信息。新的扩展正在发布，看看PostgreSQL扩展网络 (PGXN) (<https://pgxn.org/>) 可能是一个好主意，它包含了大量 PostgreSQL的扩展。

在这一章中，我们将介绍以下内容。

- 了解扩展是如何工作的
- 利用contrib模块
- 其他有用的扩展

请注意，只有最重要的扩展将被涵盖。

1. 了解扩展是如何工作的

在深入研究现有的扩展设备之前，首先看一看扩展设备是如何工作的，这是一个好主意。了解扩展机制的内部运作可能是相当有益的。

我们先来看看语法：

```
test=# \h CREATE EXTENSION
Command: CREATE EXTENSION
Description: install an extension
Syntax:
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
[ WITH ] [ SCHEMA schema_name ]
[ VERSION version ]
[ CASCADE ]
URL: https://www.postgresql.org/docs/13/sql-createextension.html
```

当你想部署一个扩展时，只需调用CREATE EXTENSION命令。它将检查该扩展并将其加载到你的数据库中。注意，扩展将被加载到数据库中，而不是整个数据库实例中。

如果我们正在加载一个扩展，我们可以决定我们要使用的模式。许多扩展可以被重新定位，以便用户可以选择使用哪种模式。然后，可以决定扩展的特定版本。通常情况下，我们不想部署最新版本的扩展，因为客户可能正在运行过时的软件。在这种情况下，能够部署系统上可用的任何版本可能是很方便的。

FROM old_version子句需要多注意一些。在过去，PostgreSQL并不支持扩展，所以很多未打包的代码仍然存在。这个选项使CREATE EXTENSION子句运行一个替代的安装脚本，将现有的对象吸收到扩展中，而不是创建新的对象。请确保SCHEMA子句指定了包含这些预先存在的对象的模式。只有当你周围有旧模块时才使用它。

最后，还有CASCADE条款。一些扩展依赖于其他扩展。CASCADE选项也会自动部署这些软件包。下面是一个例子。

```
test=# CREATE EXTENSION earthdistance;
ERROR: required extension "cube" is not installed
HINT: Use CREATE EXTENSION ... CASCADE to install required extensions too.
```

earthdistance模块实现了大圆圈距离的计算。你可能已经知道，地球上两点之间的最短距离不能以直线方式进行；相反，当从一个点飞到另一个点时，飞行员必须不断调整他们的航线以找到最快的路线。问题是，地球距离扩展依赖于立方体扩展，它允许你在球体上进行操作。

为了自动部署这种依赖关系，可以使用CASCADE子句，正如我们之前描述的那样。

```
test=# CREATE EXTENSION earthdistance CASCADE;
NOTICE: installing required extension "cube"
CREATE EXTENSION
```

在这种情况下，如NOTICE消息所示，两个扩展都将被部署。在下一节中，我们将弄清楚系统提供的是哪个扩展。

1.1 检查可用的扩展

PostgreSQL提供了各种视图，以便我们可以弄清楚哪些扩展在系统上，哪些扩展被实际部署。其中一个视图是pg_available_extensions。

```
test=# \d pg_available_extensions
View "pg_catalog.pg_available_extensions"
Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
name | name |  |  |
default_version | text |  |  |
installed_version | text | c |  |
comment | text |  |  |
```

这包含一个所有可用扩展的列表，包括它们的名称、默认版本和当前安装的版本。为了使终端用户更方便，还有一个描述，告诉我们更多关于扩展的信息

下面的列表包含了从pg_available_extensions中抽取的两行。

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pg_available_extensions LIMIT 2;
-[ RECORD 1 ]-----+
name | unaccent
default_version | 1.1
installed_version |
comment | text search dictionary that removes accents
-[ RECORD 2 ]-----+
name | tsm_system_time
default_version | 1.0
installed_version |
comment | TABLESAMPLE method which accepts time in milliseconds as a
limit
```

正如你在前面的代码块中看到的，在我的数据库中，earthdistance和plpgsql扩展都是启用的。plpgsql扩展是默认存在的，earthdistance也被添加了。这个视图的好处是，你可以迅速了解哪些东西已经安装，哪些可以安装。

然而，在某些情况下，扩展的版本不止一个。要了解更多关于版本划分的信息，请查看以下试图。

```
test=# \d pg_available_extension_versions
View "pg_catalog.pg_available_extension_versions"
Column | Type | Modifiers
-----+-----+
name | name |
version | text |
installed | boolean |
superuser | boolean |
trusted | boolean |
relocatable | boolean |
schema | name |
requires | name[] |
comment | text |
```

系统视图显示了您需要了解的有关扩展的所有信息。

这里有一些更详细的信息，如以下列表所示。

```
test=# SELECT * FROM pg_available_extension_versions LIMIT 1;
-[ RECORD 1 ]-----+
name | isn
version | 1.1
installed | f
superuser | t
trusted | t
relocatable | t
schema |
requires |
comment | data types for international product numbering standards
```

PostgreSQL也会告诉你这个扩展是否可以被重新定位，它被部署在哪个模式中，以及需要什么其他的扩展。然后，是描述扩展的注释，如前所示。

你可能想知道PostgreSQL在哪里找到所有这些关于系统中扩展的信息。假设你已经从官方的PostgreSQL RPM存储库中部署了PostgreSQL 10.0，/usr/pgsql-13/share/extension目录将包含几个文件。

```
...
-bash-4.3$ ls -l citext*
ls -l citext*
-rw-r--r--. 1 hs hs 1028 Sep 11 19:53 citext--1.0--1.1.sql
-rw-r--r--. 1 hs hs 2748 Sep 11 19:53 citext--1.1--1.2.sql
-rw-r--r--. 1 hs hs 307 Sep 11 19:53 citext--1.2--1.3.sql
-rw-r--r--. 1 hs hs 668 Sep 11 19:53 citext--1.3--1.4.sql
-rw-r--r--. 1 hs hs 2284 Sep 11 19:53 citext--1.4--1.5.sql
-rw-r--r--. 1 hs hs 13466 Sep 11 19:53 citext--1.4.sql
-rw-r--r--. 1 hs hs 158 Sep 11 19:53 citext.control
-rw-r--r--. 1 hs hs 9781 Sep 11 19:53 citext--unpackaged--1.0.sql
...
```

citext（不区分大小写的文本）扩展的默认版本是1.4，所以有一个文件叫citext--1.3.sql。除此之外，还有一些文件用于从一个版本转移到下一个版本（1.0→1.1，1.1→1.2，以此类推）。

然后是.control文件：

```
-bash-4.3$ cat citext.control
# citext extension
comment = 'data type for case-insensitive character strings'
default_version = '1.4'
module_pathname = '$libdir/citext'
relocatable = true
```

这个文件包含与扩展名相关的所有元数据；第一个条目包含注释。注意，这些内容就是我们刚才讨论的系统视图中要显示的内容。当你访问这些视图时，PostgreSQL将进入这个目录并读取所有的.control文件。然后，是默认版本和二进制文件的路径。

如果你从RPM安装一个典型的扩展，目录将是\$libdir，它在你的PostgreSQL二进制目录内。然而，如果你已经写了你自己的商业扩展，它很可能位于其他地方。

最后一项设置将告诉PostgreSQL，该扩展是否可以驻留在任何模式中，或者是否必须在一个固定的、预定义的模式中。

最后，是解压后的文件。下面是它的一个摘录。

```
...
ALTER EXTENSION citext ADD type citext;
ALTER EXTENSION citext ADD function citextin(cstring);
ALTER EXTENSION citext ADD function citextout(citext);
ALTER EXTENSION citext ADD function citextrecv(internal);
...
```

解除包装的文件将把任何现有的代码变成一个扩展。因此，在你的数据库中整合现有的代码是很重要的。在这个关于一般扩展的基本介绍之后，我们现在将研究一些额外的扩展。

2.利用contrib模块

现在我们已经看了关于扩展的理论介绍，现在是时候看一下一些最重要的扩展了。在这一节中，你将了解到作为PostgreSQL contrib模块的一部分提供给你的那些模块。当你安装PostgreSQL时，我建议你总是安装这些contrib模块，因为它们包含重要的扩展，可以真正使你的生活更容易。

在接下来的章节中，我们将引导你了解一些我认为最有趣的、对各种原因最有用的扩展（用于调试、性能调整，等等）。

2.1 使用 adminpack 模块

adminpack模块背后的想法是给管理员提供一种无需SSH访问的方法来访问文件系统。该包包含几个功能来实现这一点。

要将模块加载到数据库中，请运行以下命令：

```
test=# CREATE EXTENSION adminpack;
CREATE EXTENSION
```

adminpack模块最有趣的功能之一是检查日志文件的能力。pg_logdir_ls函数检查了日志目录并返回一个日志文件列表。

```
test=# SELECT * FROM pg_catalog.pg_logdir_ls() AS (a timestamp, b text);
ERROR: the log_filename parameter must equal 'postgresql-%Y-%m-%d_%H%M%S.log'
```

这里重要的是，log_filename参数必须根据adminpack模块的需要进行调整。如果你碰巧运行从PostgreSQL仓库下载的RPM，log_filename参数被定义为postgresql-%a，这必须被改变，以便避免错误。

进行此更改后，将返回日志文件名列表：

```
test=# SELECT * FROM pg_catalog.pg_logdir_ls() AS (a timestamp, b text);
 a | b
-----+
 2020-09-15 08:13:30 | log/postgresql-2020-09-15_081330.log
 2020-09-15 08:13:23 | log/postgresql-2020-09-15_081323.log
 2020-09-15 08:13:21 | log/postgresql-2020-09-15_081321.log
(3 rows)
```

除了这些功能外，还有一些进一步的功能是由该模块提供的。

```
test=# SELECT proname FROM pg_proc WHERE proname ~ 'pg_file_.*';
proname
-----
pg_file_write
pg_file_rename
pg_file_unlink
pg_file_read
pg_file_length
(5 rows)
```

在这里，您可以读取、写入、重命名或简单地删除文件。

当然，这些函数只能由超级用户调用。

2.2 应用布隆过滤器

从PostgreSQL 9.6开始，就可以使用扩展来即时添加索引类型。新的CREATE ACCESS METHOD命令，以及一些额外的功能，使我们有可能在飞行中创建功能齐全和有事务日志的索引类型。

bloom扩展为PostgreSQL用户提供了bloom过滤器，即前置过滤器，帮助我们尽快有效地减少数据量。布隆过滤器背后的想法是，我们可以计算一个位掩码，并将该位掩码与查询进行比较。布隆过滤器可能会产生一些误报，但它仍然会大大减少数据量。

当一个表由几百个列和几百万行组成时，这一点特别有用。数以百万计的行。用Btrees来索引几百个列是不可能的，所以Bloom filter是一个很好的选择，因为它可以让我们一次性索引一次性索引所有内容

为了了解事情是如何运作的，我们将安装该扩展。

```
test=# CREATE EXTENSION bloom;
CREATE EXTENSION
```

现在，我们需要创建一个包含各种列的表：

```
test=# CREATE TABLE t_bloom
(
    id serial,
    col1 int4 DEFAULT random() * 1000,
    col2 int4 DEFAULT random() * 1000,
    col3 int4 DEFAULT random() * 1000,
    col4 int4 DEFAULT random() * 1000,
    col5 int4 DEFAULT random() * 1000,
    col6 int4 DEFAULT random() * 1000,
    col7 int4 DEFAULT random() * 1000,
    col8 int4 DEFAULT random() * 1000,
    col9 int4 DEFAULT random() * 1000
);
CREATE TABLE
```

为了使之更容易，这些列有一个默认值，这样就可以使用一个简单的SELECT子句轻松地添加数据。

```
test=# INSERT INTO t_bloom (id)
        SELECT * FROM generate_series(1, 1000000);
INSERT 0 1000000
```

前面的查询向表中添加了 100 万行。现在，该表可以被索引：

```
test=# CREATE INDEX idx_bloom ON t_bloom
    USING bloom(col1, col2, col3, col4, col5, col6, col7, col8, col9);
CREATE INDEX
```

请注意，该索引一次包含九列。与B树相比，这些列的顺序其实并没有什么区别。

请注意，我们刚刚创建的表在没有索引的情况下大约是65MB。

该索引又增加了15 MB的存储空间：

```
test=# \di+ idx_bloom
List of relations
 Schema | Name | Type | Owner | Table | Persistence | Size | Description
-----+-----+-----+-----+-----+-----+-----+
 public | idx_bloom | index | hs | t_bloom | permanent | 15 MB |
(1 row)
```

布隆过滤器的魅力在于它可以寻找任何列的组合。

```
test=# SET max_parallel_workers_per_gather TO 0;
SET
test=# explain SELECT count(*)
   FROM t_bloom
 WHERE col4 = 454 AND col3 = 354 AND col9 = 423;
QUERY PLAN
-----
Aggregate (cost=20352.02..20352.03 rows=1 width=8)
-> Bitmap Heap Scan on t_bloom
(cost=20348.00..20352.02 rows=1 width=0)
  Recheck Cond: ((col3 = 354) AND (col4 = 454) AND (col9 = 423))
-> Bitmap Index Scan on idx_bloom
(cost=0.00..20348.00 rows=1 width=0)
  Index Cond: ((col3 = 354) AND (col4 = 454)
  AND (col9 = 423))
(5 rows)
```

到目前为止，你所看到的感觉都很特别。一个可能出现的自然问题是：为什么不总是使用布隆过滤器？原因很简单--为了使用它，数据库必须读取整个布隆过滤器。在B树的情况下，这是不需要的。

在未来，更多的索引类型可能会被添加，以确保PostgreSQL可以覆盖更多的用例。

如果你想了解更多关于bloom过滤器的信息，可以考虑阅读我们的博文：<https://www.cybertec-postgresql.com/en/trying-out-postgres-bloom-indexes/>。

2.3 部署 btree_gist 和 btree_gin

甚至还有更多与索引有关的功能可以被添加。在PostgreSQL中，有一个运算符类的概念，我们在第3章“利用索引”中讨论过这个概念。

contrib模块提供了两个扩展（即btree_gist和btree_gin），这样我们就可以为GiST和GIN索引添加B-tree功能。为什么会有这样的作用呢？GiST索引提供了B树所不支持的各种功能。其中一个功能是能够进行K-近邻（KNN）搜索。

为什么会有这种关系？想象一下，有人正在寻找昨天中午左右添加的数据。那么，那是什么时候？在某些情况下，可能很难想出边界，例如，如果有人在寻找一个价格在70欧元左右的产品。在这里，KNN可以起到拯救的作用。这里有一个例子。

```
test=# CREATE TABLE t_test (id int);
CREATE TABLE
```

需要添加一些简单数据：

```
test=# INSERT INTO t_test SELECT * FROM generate_series(1, 100000);
INSERT 0 100000
```

现在，可以添加扩展：

```
test=# CREATE EXTENSION btree_gist;
CREATE EXTENSION
```

为该列添加gist索引很容易，只要使用USING gist子句即可。注意，只有在扩展名存在的情况下，给整数列添加gist索引才有效。否则，PostgreSQL会报告说没有合适的操作符类。

```
test=# CREATE INDEX idx_id ON t_test USING gist(id);
CREATE INDEX
```

一旦部署了索引，就有可能按距离排序。

```
test=# SELECT *
  FROM t_test
 ORDER BY id <-> 100
LIMIT 6;
id
-----
100
101
99
102
98
97
(6 rows)
```

正如你所看到的，第一行是一个完全匹配。接下来的匹配已经不太精确了，而且越来越差。该查询将始终返回固定数量的行。

这里重要的是执行计划：

```
test=# explain SELECT *
  FROM t_test
 ORDER BY id <-> 100
 LIMIT 6;
QUERY PLAN

Limit (cost=0.28..0.64 rows=6 width=8)
-> Index Only Scan using idx_id on t_test
(cost=0.28..5968.28 rows=100000 width=8)
Order By: (id <-> 100)
(3 rows)
```

正如你所看到的，PostgreSQL直接进行了索引扫描，这大大加快了查询的速度。

在未来的PostgreSQL版本中，B-trees很可能也会支持KNN搜索。一个添加这个功能的补丁已经被添加到开发邮件列表中。也许它最终会进入核心版本。将KNN作为B-tree的一个功能，最终会导致标准数据类型上的GiST索引减少。

2.4 dblink——考虑逐步淘汰

使用数据库链接的愿望已经存在了很多年。然而，在世纪之交，PostgreSQL的外来数据封装器甚至还没有出现，传统的数据库链接实现也绝对没有出现。大约在这个时候，一位来自加利福尼亚的PostgreSQL开发者（Joe Conway）将dblink的概念引入到PostgreSQL中，从而开创了数据库连接的工作。虽然多年来dblink为人们提供了很好的服务，但它已经不再是最先进的了。

因此，建议我们从dblink转向更现代的SQL/MED实现（这是一个定义了外部数据与关系型数据库集成方式的规范）。postgres_fdw扩展是建立在SQL/MED之上的，它提供的不仅仅是数据库连接，因为它允许你连接到基本上任何数据源。

2.5 使用 file_fdw 获取文件

在某些情况下，从磁盘上读取一个文件并将其作为一个表暴露给PostgreSQL是有意义的。这正是你可以通过file_fdw扩展实现的。我们的想法是，有一个模块允许你从磁盘上读取数据，并使用SQL进行查询。

安装该模块的工作原理与预期一致

```
CREATE EXTENSION file_fdw;
```

现在，我们需要创建一个虚拟服务器：

```
CREATE SERVER file_server FOREIGN DATA WRAPPER file_fdw;
```

file_server是基于file_fdw扩展的外来数据包装器，它告诉PostgreSQL如何访问文件。

要把一个文件暴露为一个表，使用下面的命令。

```

CREATE FOREIGN TABLE t_passwd
(
    username text,
    passwd text,
    uid int,
    gid int,
    gecos text,
    dir text,
    shell text
) SERVER file_server
OPTIONS (format 'text', filename '/etc/passwd', header 'false', delimiter ':');

```

在这个例子中，/etc/passwd文件将被暴露。所有的字段都要被列出，数据类型也要相应地被映射。所有额外的重要信息都用OPTIONS传递给模块。在这个例子中，PostgreSQL必须知道文件的类型（文本），文件的名称和路径，以及分隔符。也可以告诉PostgreSQL是否有一个头文件。如果设置为真，第一行将被跳过，并被认为是不重要的。如果你碰巧加载了一个CSV文件，跳过标题是特别重要的。

一旦表被创建，就可以读取数据了。

```
SELECT * FROM t_passwd;
```

不出所料，PostgreSQL返回/etc/passwd的内容：

```

test=# \x
Expanded display is on.
test=# SELECT * FROM t_passwd LIMIT 1;
-[ RECORD 1 ]-----
username | root
passwd | x
uid | 0
gid | 0
gecos | root
dir | /root
shell | /bin/bash

```

当看执行计划时，你会看到PostgreSQL使用所谓的外来扫描来从文件中获取数据。

```

test=# explain (verbose true, analyze true) SELECT * FROM t_passwd;
QUERY PLAN
-----
Foreign Scan on public.t_passwd (cost=0.00..2.80 rows=18 width=168)
(actual time=0.022..0.072 rows=61 loops=1)
Output: username, passwd, uid, gid, gecos, dir, shell
Foreign File: /etc/passwd
Foreign File Size: 3484
Planning time: 0.058 ms
Execution time: 0.138 ms
(6 rows)

```

执行计划还告诉我们文件的大小等等。既然我们在谈论计划器，有一个侧面值得一提。PostgreSQL甚至会获取文件的统计数据。规划器检查文件大小并将相同成本分配给文件，就像分配给相同大小的普通PostgreSQL表一样。

2.6 使用 pageinspect 检查存储

如果你正面临存储损坏或其他一些与存储有关的问题，可能与表中的坏块有关，pageinspect扩展可能是你正在寻找的模块。我们将从创建扩展开始，如下面的例子所示。

```
test=# CREATE EXTENSION pageinspect;
CREATE EXTENSION
```

pageinspect背后的想法是为你提供一个模块，允许你在二进制层面上检查一个表。

当使用这个模块时，最重要的事情是获取一个块。

```
test=# SELECT * FROM get_raw_page('pg_class', 0);
...
```

这个函数将返回一个单一的块。在前面的例子中，它是pg_class参数中的第一个块，它是一个系统表。当然，用户可以自行选择任何其他的表。

接下来，你可以提取页头。

```
test=# \x
Expanded display is on.
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
-[ RECORD 1 ]-----
lsn | 0/8A8E310
checksum | 0
flags | 1
lower | 212
upper | 6880
special | 8192
pagesize | 8192
version | 4
prune_xid | 0
```

页头已经包含了很多关于该页的信息。如果你想了解更多，你可以调用heap_page_items函数，该函数对页面进行剖析，每一个元组返回一行。

```
test=# SELECT *
  FROM heap_page_items(get_raw_page('pg_class', 0))
  LIMIT 1;
-[ RECORD 1 ]---
tp | 1
tp_off | 49
tp_flags | 2
tp_len | 0
t xmin |
t xmax |
t_field3 |
t_ctid |
t_infomask2 |
t_infomask |
t_hoff |
t_bits |
t_oid |
t_data | ...
```

您还可以将数据拆分为各种元组：

为了读取数据，我们必须熟悉PostgreSQL的盘上格式。否则，数据可能会显得相当晦涩难懂。

`pageinspect`为所有可能的访问方法（表、索引等）提供了函数，并允许我们剖析存储，以便提供更多的细节。

2.7 使用 pg_buffercache 研究缓存

在简单介绍了pageinspect扩展之后，我们将把注意力转向pg_buffercache扩展，它允许你详细查看I/O缓存的内容。

```
test=# CREATE EXTENSION pg_buffercache;  
CREATE EXTENSION
```

`pg_buffercache` 扩展为你提供了一个包含以下字段的视图。

```
test=# \d pg_buffercache
      View "public.pg_buffercache"
 Column | Type | Collation | Nullable | Default
-----+-----+-----+-----+
bufferid | integer | | |
relfilename | oid | | |
reltablespace | oid | | |
reldatabase | oid | | |
relforknumber | smallint | | |
relblocknumber | bigint | | |
isdirty | boolean | | |
usagecount | smallint | | |
pinning_backends | integer | | |
```

bufferid字段只是一个数字；它标识了缓冲区。然后，还有relfilenode字段，它指向磁盘上的文件。如果我们想查询一个文件属于哪个表，我们可以查看pg_class模块，它也包含一个叫做relfilenode的字段。然后，还有reldatabase和reltablespace字段。注意，所有的字段都被定义为oid类型，所以为了以更有用的方式提取数据，有必要将系统表连接起来。

`relforknumber`字段告诉我们该表的哪一部分被缓存了。它可能是堆，自由空间图，或其他一些组件，如可见性图。在未来，肯定会有更多类型的关系分叉。下一个字段，`relblocknumber`，告诉我们哪个块已经被缓存了。最后是`isdirty`标志，它告诉我们一个区块已经被修改了，以及关于使用计数器和钉住该区块的后端数量。

如果你想了解`pg_buffercache`扩展的意义，增加额外的信息很重要。要弄清楚哪个数据库使用缓存最多，下面的查询可能会有帮助。

```
test=# SELECT datname,
  count(*),
  count(*) FILTER (WHERE isdirty = true) AS dirty
FROM pg_buffercache AS b, pg_database AS d
WHERE d.oid = b.reldatabase
GROUP BY ROLLUP (1);
datname | count | dirty
-----+-----+
abc   | 132  | 1
postgres | 30  | 0
test   | 11975 | 53
      | 12137 | 54
(4 rows)
```

在这种情况下，`pg_database`扩展必须被连接。我们可以看到，`oid`是连接的标准，这对刚接触PostgreSQL的人来说可能不是很明显。

有时，我们可能想知道数据库中哪些与我们相连的块被缓存了。下面是它的原理。

```
test=# SELECT relname,
  relkind,
  count(*),
  count(*) FILTER (WHERE isdirty = true) AS dirty
FROM pg_buffercache AS b, pg_database AS d, pg_class AS c
WHERE d.oid = b.reldatabase
AND c.relfilenode = b.relfilenode
AND datname = 'test'
GROUP BY 1, 2
ORDER BY 3 DESC
LIMIT 7;
relname | relkind| count| dirty
-----+-----+-----+
t_bloom | r | 8338 | 0
idx_bloom | i | 1962 | 0
idx_id | i | 549 | 0
t_test | r | 445 | 0
pg_statistic | r | 90 | 0
pg_depend | r | 60 | 0
pg_depend_reference_index | i | 34 | 0
(7 rows)
```

在这种情况下，我们过滤了当前的数据库，并与包含对象列表的`pg_class`模块连接。`relkind`列特别值得注意：`r`指的是表（关系），`i`指的是索引。这告诉我们，我们正在看哪个对象。

2.8 使用 pgcrypto 加密数据

在整个contrib模块部分，最强大的模块之一是pgcrypto。它最初是由Skype的一个系统管理员编写的，提供了无数的功能，使我们能够加密和解密数据。

它提供了对称和非对称加密的功能。由于提供了大量的功能，绝对建议查看文档页面：<https://www.postgresql.org/docs/current/static/pgcrypto.html>。

由于本章的范围有限，我们不可能挖掘pgcrypto模块的所有细节。

2.9 使用 pg_prewarm 预热缓存

当PostgreSQL正常运行时，它试图对重要数据进行缓存。shared_buffers这个变量很重要，因为它定义了由PostgreSQL管理的缓存的大小。现在的问题是这样的：如果你重新启动数据库服务器，由PostgreSQL管理的缓存就会丢失。也许操作系统还有一些数据可以减少对磁盘等待的影响，但在很多情况下，这是不够的。解决这个问题的方法叫做pg_prewarm扩展。现在让我们安装pg_prewarm，看看我们能用它做什么。

```
test=# CREATE EXTENSION pg_prewarm;
CREATE EXTENSION
```

这个扩展部署了一个函数，允许我们在需要时明确地预热缓存。该列表显示了与该扩展相关的函数。

```
test=# \x
Expanded display is on.
test=# \df *prewa*
List of functions
-[ RECORD 1 ]
Schema | public
Name | autoprewarm_dump_now
Result data type | bigint
Argument data types |
Type | func
-[ RECORD 2 ]
Schema | public
Name | autoprewarm_start_worker
Result data type | void
Argument data types |
Type | func
-[ RECORD 3 ]
Schema | public
Name | pg_prewarm
Result data type | bigint
Argument data types | regclass, mode text DEFAULT 'buffer'::text,
fork text DEFAULT 'main'::text,
first_block bigint DEFAULT NULL::bigint,
last_block bigint DEFAULT NULL::bigint
Type | func
```

调用 pg_prewarm 扩展最简单和最常见的方法是要求它缓存整个对象：

```
test=# SELECT pg_prewarm('t_test');
pg_prewarm
-----
443
(1 row)
```

请注意，如果表太大以至于无法放入缓存，则只有部分表会保留在缓存中，这在大多数情况下都很好。

该函数返回被该函数调用处理的8KB块的数量。如果你不想缓存一个对象的所有块，你也可以在表中选择一个特定的范围。

在下面的例子中，我们可以看到10到30块被缓存在主分叉中。

```
test=# SELECT pg_prewarm('t_test', 'buffer', 'main', 10, 30);
pg_prewarm
-----
21
(1 row)
```

在这里，很明显缓存了 21 个块。

2.10 使用 pg_stat_statements 检查性能

pg_stat_statements是目前最重要的contrib模块。它应该总是被启用，并且是为了提供卓越的性能数据。如果没有pg_stat_statements模块，真的很难追踪到性能问题。

2.11 使用 pgstattuple 检查存储

有时，PostgreSQL中的表可能会增长得不成比例。表增长过多的技术术语是表膨胀。现在出现的问题是：哪些表已经膨胀了，有多少膨胀了？pgstattuple扩展将帮助我们回答这些问题。

```
test=# CREATE EXTENSION pgstattuple;
CREATE EXTENSION
```

正如我们之前所说，该模块部署了几个函数。在pgstattuple扩展的情况下，这些函数返回一个由复合类型组成的行。因此，必须在FROM子句中调用该函数，以确保有一个可读的结果。

```
test=# \x
Expanded display is on.
test=# SELECT * FROM pgstattuple('t_test');
-[ RECORD 1 ]+
-----+-----+
table_len | 3629056
tuple_count | 100000
tuple_len | 2800000
tuple_percent | 77.16
dead_tuple_count | 0
dead_tuple_len | 0
dead_tuple_percent | 0
free_space | 16652
free_percent | 0.46
```

在这个例子中，用于测试的表似乎处于一个相当好的状态：表的大小为3.6MB，不包含任何死行。自由空间也是有限的。如果对你的表的访问因表的膨胀而变慢，那么这意味着死行的数量和自由空间的数量将不成比例地增长。一些自由空间和少量的死行是正常的；但是，如果表已经增长到大部分由死行和自由空间组成，就需要采取果断的行动，使情况再次得到控制。

pgstattuple扩展也提供了一个函数，我们可以用它来检查索引。

```
test=# CREATE INDEX idx_id ON t_test (id);
CREATE INDEX
```

pgstattindex函数返回了很多关于我们要检查的索引的信息。

```
test=# SELECT * FROM pgstattindex('idx_id');
-[ RECORD 1 ]
-----+
version | 2
tree_level | 1
index_size | 2260992
root_block_no | 3
internal_pages | 1
leaf_pages | 274
empty_pages | 0
deleted_pages | 0
avg_leaf_density | 89.83
leaf_fragmentation | 0
```

我们的指数是相当密集的（89%）。这是一个好的迹象。索引的默认FILLFACTOR设置是90%，所以接近90%的数值表明索引非常好。

有时，你不想检查一个表；相反，你想检查所有的表，或者只检查一个模式中的所有表。如何实现这一点呢？通常情况下，你想处理的对象的列表在FROM子句中。然而，在我的例子中，函数已经在FROM子句中了，那么我们怎样才能使PostgreSQL在表的列表中循环呢？答案是使用一个LATERAL连接。

请记住，pgstattuple必须要读取整个对象。如果我们的数据库很大，它可能需要相当长的时间来处理。因此，存储我们刚刚看到的查询结果可能是一个好主意，这样我们就可以彻底检查它们，而不必一次又一次地重新运行该查询

2.12 使用 pg_trgm 进行模糊搜索

pg_trgm模块允许你进行模糊搜索。这个模块在第三章 "利用索引" 中讨论过。

2.13 使用 postgres_fdw 连接到远程服务器

数据并不总是只在一个地方可用。更多的时候，数据分布在基础设施中，可能驻留在不同地方的数据必须被整合。

解决这个问题的方法就是SQL/MED标准所定义的外来数据包装器。

在这一节中，我们将讨论postgres_fdw扩展。它是一个允许我们从PostgreSQL数据源动态获取数据的模块。我们需要做的第一件事是部署外部数据包装器。

```
test=# \h CREATE FOREIGN DATA WRAPPER
Command: CREATE FOREIGN DATA WRAPPER
Description: define a new foreign-data wrapper
Syntax:
CREATE FOREIGN DATA WRAPPER name
[ HANDLER handler_function | NO HANDLER ]
[ VALIDATOR validator_function | NO VALIDATOR ]
[ OPTIONS ( option 'value' [, ... ] ) ]
URL: https://www.postgresql.org/docs/13/sql-createforeigndatawrapper.html
```

幸运的是，CREATE FOREIGN DATA WRAPPER命令隐藏在一个扩展中。它可以很容易地使用正常程序安装，如下所示。

```
test=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
```

现在，必须要定义一个虚拟服务器。它将指向另一台主机，并告诉PostgreSQL从哪里获得数据。在数据的最后，PostgreSQL必须建立一个完整的连接字符串--服务器数据是PostgreSQL首先要知道的东西。用户信息将在以后添加。服务器将只包含主机、端口，等等。CREATE SERVER的语法如下

```
test=# \h CREATE SERVER
Command: CREATE SERVER
Description: define a new foreign server
Syntax:
CREATE SERVER [ IF NOT EXISTS ] server_name [ TYPE 'server_type' ] [ VERSION
'server_version' ]
FOREIGN DATA WRAPPER fdw_name
[ OPTIONS ( option 'value' [, ...] ) ]
URL: https://www.postgresql.org/docs/13/sql-createserver.html
```

为了理解这一点，我们将在同一主机上创建第二个数据库，并创建一个服务器。

```
[hs@zenbook~]$ createdb customer
[hs@zenbook~]$ psql customer
customer=# CREATE TABLE t_customer (id int, name text);
CREATE TABLE
customer=# CREATE TABLE t_company (
country text,
name text,
active text
);
CREATE TABLE
customer=# \d
List of relations
 Schema | Name | Type | Owner
-----+-----+-----+
 public | t_company | table |
 hs    | t_customer | table | hs
(2 rows)
```

现在，应该将服务器添加到标准测试数据库：

```
test=# CREATE SERVER customer_server
FOREIGN DATA WRAPPER postgres_fdw
OPTIONS (host 'localhost', dbname 'customer', port '5432');
CREATE SERVER
```

请注意，所有的重要信息都是以OPTIONS子句的形式存储的。这一点有些重要，因为它给了用户很大的灵活性。有许多不同的外部数据包装器，每个包装器都需要不同的选项。

一旦服务器被定义，就到了映射用户的时候了。如果我们从一个服务器连接到另一个服务器，我们在两个地方的用户可能不是同一个人。因此，外部数据封装器需要用户定义实际的用户映射，如下所示。

```
test=# \h CREATE USER MAPPING
Command: CREATE USER MAPPING
Description: define a new mapping of a user to a foreign server
Syntax:
CREATE USER MAPPING [ IF NOT EXISTS ] FOR { user_name | USER | CURRENT_USER |
PUBLIC }
    SERVER server_name
[ OPTIONS ( option 'value' [ , ... ] ) ]
URL: https://www.postgresql.org/docs/13/sql-createusermapping.html
```

语法非常简单，可以很容易地使用：

```
test=# CREATE USER MAPPING
FOR CURRENT_USER SERVER customer_server
OPTIONS (user 'hs', password 'abc');
CREATE USER MAPPING
```

同样，所有的重要信息都隐藏在OPTIONS子句中。根据外部数据封装器的类型，选项列表会有所不同。请注意，我们必须在这里使用适当的用户数据，这对我们的设置是有效的。在这种情况下，我们将简单地使用本地用户。

一旦基础设施到位，我们就可以创建外域表了。创建外域表的语法与我们创建普通本地表的方法非常相似。所有的列都必须被列出，包括它们的数据类型。

```
test=# CREATE FOREIGN TABLE f_customer (id int, name text)
SERVER customer_server
OPTIONS (schema_name 'public', table_name 't_customer');
CREATE FOREIGN TABLE
```

所有的列都被列出，就像在普通的CREATE TABLE子句中的情况。这里的特殊之处在于，外域表指向了远程端的一个表。模式的名称和表的名称必须在OPTIONS子句中指定。

一旦它被创建，该表就可以被使用

```
test=# SELECT * FROM f_customer ;
 id | name
-----+
(0 rows)
```

要检查PostgreSQL内部做了什么，最好是运行带有分析参数的EXPLAIN子句。它将揭示一些关于服务器中真正发生的事情的信息。

```

test=# EXPLAIN (analyze true, verbose true)
SELECT * FROM f_customer ;
QUERY PLAN
-----
Foreign Scan on public.f_customer
(cost=100.00..150.95 rows=1365 width=36)
(actual time=0.221..0.221 rows=0 loops=1)
Output: id, name
Remote SQL: SELECT id, name FROM public.t_customer
Planning time: 0.067 ms
Execution time: 0.451 ms
(5 rows)

```

这里的重要部分是远程SQL。外部数据封装器将向对方发送查询，并尽可能少地获取数据，因为尽可能多的限制条件在远程端执行，以确保没有太多的数据被本地处理。过滤条件、连接、甚至聚合都可以在远程执行（从PostgreSQL 10.0开始）。

虽然CREATE FOREIGN TABLE子句肯定是个好东西，但反复列出所有这些列可能是相当麻烦的。

解决这个问题的方法是IMPORT子句。这使我们能够快速而方便地将整个模式导入本地数据库，并创建外域表。

```

test=# \h IMPORT
Command: IMPORT FOREIGN SCHEMA
Description: import table definitions from a foreign server
Syntax:
IMPORT FOREIGN SCHEMA remote_schema
[ { LIMIT TO | EXCEPT } ( table_name [, ...] ) ]
FROM SERVER server_name
INTO local_schema
[ OPTIONS ( option 'value' [, ...] ) ]
URL: https://www.postgresql.org/docs/13/sql-importforeignschema.html

```

IMPORT使我们能够很容易地连接大型表格集。由于所有的信息都是直接从远程数据源获取的，所以它也减少了错别字和错误的机会。

下面是它的工作原理。

```

test=# IMPORT FOREIGN SCHEMA public
  FROM SERVER customer_server INTO public;
IMPORT FOREIGN SCHEMA

```

在这种情况下，之前在公共模式中创建的所有表都被直接链接。我们可以看到，所有的远程表现在都可以使用。

```

test=# \det
List of foreign tables
Schema | Table | Server
-----+-----+
public | f_customer | customer_server
public | t_company | customer_server
public | t_customer | customer_server
(3 rows)

```

\det列出了所有的外来表，如前面的代码所示。

2.13.1 处理错误和拼写错误

创建外域表其实并不难，然而，有时会发生人们犯错的情况，或者是已经使用的密码简单地改变了。为了处理这些问题，PostgreSQL提供了两个命令。ALTER SERVER和ALTER USER MAPPING。

ALTER SERVER允许你修改一个服务器。下面是它的语法

```
test=# \h ALTER SERVER
Command: ALTER SERVER
Description: change the definition of a foreign server
Syntax:
ALTER SERVER name [ VERSION 'new_version' ]
[ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ...] ) ]
ALTER SERVER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SERVER name RENAME TO new_name
URL: https://www.postgresql.org/docs/13/sql-alterserver.html
```

我们可以使用这个命令来添加和删除特定服务器的选项，如果我们忘记了什么，这是件好事。

要修改用户信息，我们也可以改变用户的映射。

```
test=# \h ALTER USER MAPPING
Command: ALTER USER MAPPING
Description: change the definition of a user mapping
Syntax:
ALTER USER MAPPING FOR { user_name | USER | CURRENT_USER | SESSION_USER | PUBLIC
}
    SERVER server_name
OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ...] )
URL: https://www.postgresql.org/docs/13/sql-alterusermapping.html
```

SQL/MED接口正在不断地被改进，在写这篇文章的时候，功能正在被添加。在未来，甚至更多的优化将进入核心，使SQL/MED接口成为提高可扩展性的良好选择。

到目前为止，你已经学会了如何使用外来数据包装器。现在让我们来看看一些更有用的扩展。

3.其他有用的扩展

到目前为止，我们所描述的扩展都是PostgreSQL contrib包的一部分，它是作为PostgreSQL源代码的一部分被运送的。然而，我们在这里看到的包并不是PostgreSQL社区中唯一可用的包。还有很多包允许我们做各种各样的事情。

不幸的是，这一章太短了，无法深入研究目前存在的所有东西。模块的数量每天都在增长，不可能涵盖所有的模块。因此，我只想指出我认为最重要的那些。

PostGIS (<http://postgis.net/>) 是开源世界中的地理信息系统 (GIS) 数据库接口。它已被全球采用，是关系型开源数据库世界的事例标准。它是一个专业和极其强大的解决方案。

如果你正在寻找地理空间路由，pgRouting就是你可能正在寻找的东西。它提供了各种算法，你可以用它来寻找地点之间的最佳连接，并在PostgreSQL的基础上工作。

在本章中，我们已经了解了postgres_fdw扩展，它允许我们连接到其他一些PostgreSQL数据库。周围还有许多外部数据包装器。其中最著名和最专业的是oracle_fdw扩展。它允许你与Oracle集成，并通过电线获取数据，这可以通过postgres_fdw扩展来完成。

在某些情况下，你也可能对用pg_crash (https://github.com/cybertec-postgresql/pg_crash) 测试基础设施的稳定性感兴趣。这个想法是要有一个模块来不断地崩溃你的数据库。

pg_crash模块是测试和调试连接池的绝佳选择，它允许你重新连接到一个失败的数据库。pg_crash模块会周期性地疯狂运行，杀死数据库会话或破坏内存。它是长期测试的理想选择。

4. 总结

在这一章中，我们了解了一些最有前途的模块，这些模块是随PostgreSQL标准发布版一起提供的。这些模块相当多样化，提供了从数据库连接到区分大小写的文本和模块，以便我们可以检查服务器。然而，在本节中，你已经了解了周围最重要的模块。这将帮助你部署更伟大的数据库设置。

现在我们已经处理了扩展问题，在下一章，我们将把注意力转移到迁移上。在那里，我们将学习如何以最简单的方式迁移到PostgreSQL。

排除PostgreSQL的故障

1.接近一个未知的数据库

2.检查pg_stat_activity

- 2.1 查询 pg_stat_activity
- 2.2 处理 Hibernate 语句
- 2.3 找出查询的来源

3.检查慢的查询

- 3.1 检查单个查询
- 3.2 使用 perf 进行更深入的挖掘

4.检查日志

5.检查丢失的索引

6.检查内存和I/O

7.了解值得注意的错误情况

- 7.1 面对堵塞腐败
- 7.2 了解检查点消息
- 7.3 管理损坏的数据页
- 7.4 粗心的连接管理
- 7.5 对抗表的膨胀

8.总结

9.问题

在第11章 "决定有用的扩展" 中，我们了解了一些被广泛采用的有用的扩展，它们可以给你的部署带来真正的提升。作为后续，现在将向你介绍PostgreSQL的故障排除。其目的是给你一个系统的方法来检查和修复你的系统，以提高性能和避免常见的陷阱。有一个系统的方法肯定会得到回报。

在这一章中，将包括以下主题。

- 接近一个未知的数据库
- 检查pg_stat_activity
- 检查慢的查询
- 检查日志
- 检查丢失的索引
- 检查内存和I/O
- 了解值得注意的错误情况

请记住，许多事情都可能出错，因此，对数据库进行专业监测是很重要的。要弄清楚什么地方出了问题，你必须以专业的方式来研究这个系统。

1.接近一个未知的数据库

如果你碰巧要管理一个大规模的系统，你可能不知道系统到底在做什么。管理数以百计的系统意味着你不会知道每个系统发生了什么。当涉及到故障排除时，最重要的事情可以归结为一个词：数据。如果没有足够的数据，就没有办法解决事情。因此，排除故障的第一步就是一定要设置一个监控工具，比如pgwatch2（可在<https://www.cybertec-postgresql.com/en/products/pgwatch2/>），它可以帮助你对数据库服务器有一些了解。一旦报告工具告诉你一个值得检查的情况，就意味着它已经被证明是有用的，可以有组织地接近系统。

2. 检查pg_stat_activity

首先，让我们检查pg_stat_statements的内容，并回答以下问题。

- 目前你的系统上有多少个并发的查询正在执行？
- 你是否看到类似类型的查询一直显示在查询栏中？
- 你是否看到有的查询已经运行了很长时间？
- 是否有任何锁没有被授予？
- 你是否看到有来自可疑主机的连接？

应该首先检查pg_stat_activity视图，因为它可以让我们了解系统上正在发生什么。当然，图形化监控应该给你一个系统的第一印象。然而，在一天结束的时候，它真正归结为在服务器上实际运行的查询。因此，由pg_stat_activity提供的对系统的良好概述对于追踪问题是十分重要的。

为了让你更轻松，我汇编了一些我认为对尽快发现问题有用的查询。

2.1 查询 pg_stat_activity

下面的查询显示你的数据库目前正在执行多少个查询。

```
test=# SELECT datname,
  count(*) AS open,
  count(*) FILTER (WHERE state = 'active') AS active,
  count(*) FILTER (WHERE state = 'idle') AS idle,
  count(*) FILTER (WHERE state = 'idle in transaction')
  AS idle_in_trans
FROM pg_stat_activity
WHERE backend_type = 'client backend'
GROUP BY ROLLUP(1);
datname | open | active | idle | idle_in_trans
-----+-----+-----+-----+
test  | 2  | 1  | 0  | 1
| 2  | 1  | 0  | 1
(2 rows)
```

为了在同一个屏幕上显示尽可能多的信息，使用了部分聚合。我们可以看到活跃、闲置和闲置在交易中的查询。如果我们可以看到大量的闲置在事务中的查询，那么肯定要深入挖掘，以便弄清楚这些事务被保持了多长时间。下面的列表显示了可以找到多长时间的交易。

```
test=# SELECT pid, xact_start, now() - xact_start AS duration
  FROM pg_stat_activity
 WHERE state LIKE '%transaction%'
 ORDER BY 3 DESC;
pid | xact_start | duration
-----+-----+
19758 | 2020-09-26 20:27:08.168554+01 | 22:12:10.194363
(1 row)
```

前面列表中的事务已经持续了22个多小时。现在的主要问题是：一个事务怎么可能开放这么长时间？在大多数应用中，一个需要这么长时间的事务是非常可疑的，而且可能非常危险的。危险从何而来？正如我们在本书前面所学到的，只有在没有事务可以看到死行的情况下，VACUUM命令才能清理这些死行。现在，如果一个事务持续开放几个小时甚至几天，VACUUM命令就不能产生有用的结果，这将导致

表的膨胀。

因此，我们强烈建议确保长的事务被监控或杀死，以防它们变得太长。从9.6版本开始，PostgreSQL有一个叫做快照过期的功能，它允许我们在快照过期的情况下终止长事务。

检查是否有任何长期运行的查询，也是一个好主意。

```
test=# SELECT now() - query_start AS duration, datname, query
  FROM pg_stat_activity
 WHERE state = 'active'
 ORDER BY 1 DESC;
duration | datname | query
-----+-----+
00:00:38.814526 | dev | SELECT pg_sleep(10000);
00:00:00 | test | SELECT now() - query_start AS duration,
datname, query
  FROM pg_stat_activity
 WHERE state = 'active'
 ORDER BY 1 DESC;
(2 rows)
```

在这种情况下，所有活跃的查询都被抽取出来，报表计算出每个查询已经活跃了多长时间。通常情况下，我们会看到类似的查询排在前面，这可以给我们提供一些有价值的线索，说明我们的系统正在发生什么。

2.2 处理 Hibernate 语句

许多对象关系映射 (ORM)，例如 Hibernate，会生成非常长的 SQL 语句。问题在于：pg_stat_activity 只会存储系统视图中查询的前 1024 个字节。其余部分被截断。对于 Hibernate 等 ORM 生成的长查询，查询在感兴趣的部分 (FROM 子句等) 真正开始之前被切断。

这个问题的解决方案是在 postgresql.conf 文件中设置一个配置参数：

```
test=# SHOW track_activity_query_size;
track_activity_query_size
-----
1kB
(1 row)
```

如果我们把这个参数增加到一个合理的高值（也许是32,768），并重新启动PostgreSQL，那么我们将能够看到更长的查询，并能够更容易地发现问题。

2.3 找出查询的来源

在检查pg_stat_activity时，有一些字段会告诉我们一个查询来自哪里。

```
client_addr | inet |
client_hostname | text |
client_port | integer |
```

这些字段将包含IP地址和主机名（如果已配置）。但是，如果每个应用程序都从同一个IP发送请求，例如，所有的应用程序都驻扎在同一个应用服务器上，会发生什么？我们将很难看到哪个应用程序产生了某个查询。

解决这个问题的办法是要求开发人员设置一个application_name变量。

```
test=# SHOW application_name ;
application_name

-----
pgsql
(1 row)
test=# SET application_name TO 'some_name';
SET
test=# SHOW application_name ;
application_name

-----
some_name
(1 row)
```

如果人们合作，`application_name`变量将显示在系统视图中，使人们更容易看到查询的来源。`application_name`变量也可以作为连接字符串的一部分来设置。在下一步，我们将尝试弄清与缓慢查询有关的一切。

3. 检查慢的查询

在检查了`pg_stat_activity`之后，看一下缓慢的、耗时的查询是有意义的。基本上，有两种方法可以解决这个问题：

- 在日志中寻找单个的慢速查询
- 找出耗时过长的查询类型

寻找单一的、缓慢的查询是性能调整的经典方法。通过设置`log_min_duration_statement`变量到一个期望的阈值，PostgreSQL将开始为每个超过这个阈值的查询写一个日志行。默认情况下，慢查询日志是关闭的，如下所示。

```
test=# SHOW log_min_duration_statement;
log_min_duration_statement

-----
-1
(1 row)
```

然而，将这个变量设置为一个合理的值是非常合理的。根据你的工作量，所需的时间当然可能不同。在许多情况下，期望值可能因数据库而异。因此，也可以以更精细的方式使用该变量。

在许多情况下，所需的值可能因数据库而异。因此，也可以以更精细的方式使用该变量。

```
test=# ALTER DATABASE test SET log_min_duration_statement TO 10000;
ALTER DATABASE
```

如果你的数据库面临不同的工作负荷，只为某个数据库设置该参数是非常有意义的。

当使用慢速查询日志时，必须考虑一个重要的因素--许多较小的查询可能会导致更大的负载，而不仅仅是少数几个运行缓慢的查询。当然，意识到个别慢速查询总是有意义的，但有时，这些查询并不是问题所在。

考虑下面的例子：在你的系统上，执行了100万个查询，每个需要500毫秒，同时还有一些分析性查询，每个运行了几毫秒。显然，真正的问题永远不会出现在慢速查询日志中，而每一次数据导出、每一次索引创建和每一次批量加载（无论如何在大多数情况下是无法避免的）都会在日志中出现垃圾信息，给我们指出错误的方向。

因此，我个人的建议是，使用慢查询日志，但要小心使用，谨慎使用。但最重要的是，要注意我们真正地在测量什么。

在我看来，更好的方法是更深入地使用pg_stat_statements视图。它将提供汇总的信息，而不仅仅是关于单个查询的信息。本书前面已经讨论了pg_stat_statements视图。然而，这个模块的重要性怎么强调都不为过。

3.1 检查单个查询

有时，慢的查询被识别出来，但我们仍然不知道到底发生了什么。下一步当然是检查查询的执行计划，看看会发生什么。识别计划中那些导致不良运行时间的关键操作是相当简单的。试着使用下面的检查表。

- 试着看看在计划的什么地方，时间开始急剧上升。
- 检查是否有缺失的索引（性能不佳的主要原因之一）。
- 使用EXPLAIN子句（缓冲区为真，分析为真，以此类推），看看你的查询是否使用了太多的缓冲区。
- 打开track_io_timing参数，弄清是I/O问题还是CPU问题（明确检查是否有随机I/O发生）。
- 寻找不正确的估计，并尝试修复它们。
- 寻找那些执行频率过高的存储过程。
- 试着找出其中一些是否可以被标记为STABLE或IMMUTABLE，如果可以的话。

注意，pg_stat_statements不考虑解析时间，所以如果你的查询非常长（比如查询字符串），那么pg_stat_statements可能会有一点误导。

3.2 使用 perf 进行更深入的挖掘

在大多数情况下，通过这个微小的检查表工作将帮助你以相当快和有效的方式追踪大多数的问题。然而，即使是从数据库引擎中提取的信息有时也是不够的。

perf工具是一个用于Linux的分析工具，它允许你直接看到哪些C函数在你的系统上引起问题。通常情况下，perf不是默认安装的，所以建议你安装它。要在你的服务器上使用perf，只需以root身份登录并运行以下命令。

```
perf top
```

屏幕每隔几秒钟就会刷新一次，你将有机会看到现场发生的情况。下面的列表向你展示了一个标准的、只读的基准可能是什么样子。

```
Samples: 164K of event 'cycles:ppp', Event count (approx.): 109789128766
Overhead Shared Object Symbol
 3.10% postgres [.] AllocSetAlloc
 1.99% postgres [.] SearchCatCache
 1.51% postgres [.] base_yyparse
 1.42% postgres [.] hash_search_with_hash_value
 1.27% libc-2.22.so [.] vfprintf
 1.13% libc-2.22.so [.] _int_malloc
 0.87% postgres [.] palloc
 0.74% postgres [.] MemoryContextAllocZeroAligned
 0.66% libc-2.22.so [.] __strcmp_sse2_unaligned
```

```
0.66% [kernel] [k] _raw_spin_lock_irqsave
0.66% postgres [.] _bt_COMPARE
0.63% [kernel] [k] __fget_light
0.62% libc-2.22.so [.] strlen
```

你可以看到，在我们的样本中，没有任何一个函数占用过多的CPU时间，这告诉我们，系统是很好的。

然而，情况可能并不总是这样的。有一个叫做自旋锁争夺的问题是很常见的。自旋锁被PostgreSQL核心用来同步诸如缓冲区访问等事情。自旋锁是现代CPU提供的一个功能，以避免操作系统对小操作（如增加一个数字）的互动。如果你认为你可能面临自旋锁的争夺，其症状如下。

- 一个真正的高CPU负载。
- 难以置信的低吞吐量（通常需要几毫秒的查询突然需要几秒钟）。
- I/O异常低，因为CPU正忙于交易锁。

在许多情况下，自旋锁争用是突然发生的。你的系统刚刚还好的，突然间，负载上升了，吞吐量像石头一样下降了。perf top命令会显示，大部分时间都花在一个叫s_lock的C函数上。如果是这种情况，你应该尝试做以下工作。

```
huge_pages = try # on, off, or try
```

将huge_pages从try改为off。在操作系统层面上完全关闭巨大页面可能是一个好主意。一般来说，似乎有些内核比其他内核更容易产生这类问题。红帽2.6.32系列似乎特别糟糕（注意，我在这里使用了似乎这个词）。

如果你正在使用PostGIS，perf工具也很有趣。如果列表中的顶级函数都是与GIS有关的（比如，来自一些底层库），你就知道问题很可能不是来自PostgreSQL的不良调整，而只是与需要时间完成的昂贵操作有关。

4. 检查日志

如果你的系统闻到了麻烦的味道，检查日志以了解发生了什么是有意义的。重要的一点是：并不是所有的日志条目都是同样创建的。PostgreSQL有一个从DEBUG到PANIC的日志条目的层次结构。

对于管理员来说，以下三个错误级别是非常重要的。

- ERROR
- FATAL
- PANIC

ERROR用于处理诸如语法错误、权限相关问题等问题。你的日志将总是包含错误信息。关键的因素是--某种类型的错误出现的频率如何？产生数以百万计的语法错误当然不是运行数据库服务器的理想策略。

FATAL比ERROR更可怕；你会看到诸如无法为共享内存分配内存或意外的walreceiver状态等信息。换句话说，这些错误信息已经非常可怕了，会告诉你事情正在出错。

最后，是PANIC。如果你碰到这种信息，你就知道事情真的非常非常不对劲。PANIC的典型例子是锁表被破坏，或者创建了太多的信号灯。这些都会导致关机。在下一节中，你将了解丢失索引的情况。

5. 检查丢失的索引

一旦我们完成了前三个步骤，重要的是看一下总体的性能。正如我在本书中一直强调的那样，缺失的索引是造成超差性能的全部原因。所以，每当我们面对一个缓慢的系统时，建议我们检查是否有缺失的索引，并部署任何需要的东西。

通常情况下，客户要求我们优化RAID级别，调整内核，或者其他一些花哨的东西。在现实中，这些复杂的要求往往可以归结为少量的索引丢失。根据我的判断，花一些额外的时间来检查所有需要的索引是否在那里总是有意义的。检查缺失的索引既不难也不费时，所以应该一直这样做，不管你面临什么样的性能问题。

这是我最喜欢的查询，以获得对可能缺少索引的地方的印象。

```
SELECT schemaname, relname, seq_scan, seq_tup_read,
       idx_scan, seq_tup_read / seq_scan AS avg
  FROM pg_stat_user_tables
 WHERE seq_scan > 0
 ORDER BY seq_tup_read DESC
LIMIT 20;
```

试着找到经常被扫描的大表（平均价值高）。这些表通常会排在前面。

6. 检查内存和I/O

一旦我们完成了寻找丢失的索引，我们就可以检查内存和I/O。为了弄清楚发生了什么，激活track_io_timing是有意义的。如果它是开启的，PostgreSQL将收集关于磁盘等待时间的信息并将其呈现给你。

通常，客户问的主要问题是：如果我们增加更多的磁盘，它是否会更快？有可能猜到会发生什么，但一般来说，测量是更好、更有用的策略。启用track_io_timing将帮助你收集数据，以真正搞清楚这个问题。

PostgreSQL以各种方式暴露了磁盘等待时间。检查事情的一个方法是看一下pg_stat_database。

```
test=# \d pg_stat_database
      View "pg_catalog.pg_stat_database"
 Column | Type | Modifiers
-----+-----+
 datid | oid |
 datname | name |
 ...
 conflicts | bigint |
 temp_files | bigint |
 temp_bytes | bigint |
 ...
 blk_read_time | double precision |
 blk_write_time | double precision |
```

请注意，在最后有两个字段 - blk_read_time和blk_write_time。它们将告诉我们PostgreSQL花了多少时间来等待操作系统的响应。请注意，我们在这里不是真正的测量磁盘等待时间，而是测量操作系统返回数据所需的时间。

如果操作系统产生缓存命中，这个时间会相当低。如果操作系统要做非常讨厌的随机I/O，我们会看到一个区块甚至需要几毫秒。

在很多情况下，当temp_files和temp_bytes显示高数字时，就会出现高的blk_read_time和blk_write_time。另外，在许多情况下，这指向一个糟糕的work_mem设置或一个糟糕的maintain_work_mem设置。记住这一点：如果PostgreSQL不能在内存中做事情，它必须溢出到磁盘上。你可以使用temp_files操作来检测这一点。只要有temp_files，就有可能出现讨厌的磁盘等待时间。

虽然在每个数据库层面上的全局视图是有意义的，但它并不能产生关于麻烦的真正来源的深入信息。通常情况下，只有少数查询是导致性能不佳的原因。发现这些问题的方法是使用pg_stat_statements。

```
test=# \d pg_stat_statements
View "public.pg_stat_statements"
Column | Type | Modifiers
-----+-----+-----+
...
query | text |
calls | bigint |
total_time | double precision |
...
temp_blk_reads | bigint |
temp_blk_written | bigint |
blk_read_time | double precision |
blk_write_time | double precision |
```

你将能够看到，在每个查询的基础上，是否有磁盘等待。重要的部分是blk_time值和total_time的组合。这个比例才是最重要的。一般来说，一个显示超过30%的磁盘等待的查询可以被看作是严重的I/O绑定。

一旦我们完成了对PostgreSQL系统表的检查，检查Linux上的vmstat命令告诉我们的信息是有意义的。或者，我们可以使用iostat命令。

```
[hs@zenbook ~]$ vmstat 2
procs -----memory----- --swap-- -----io---- -system-- -----cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa st
0 0 367088 199488 96 2320388 0 2 83 96 106 156 16 6 78 0 0
0 0 367088 198140 96 2320504 0 0 0 10 595 2624 3 1 96 0 0
0 0 367088 191448 96 2320964 0 0 0 8 920 2957 8 2 90 0 0
```

在做数据库工作时，我们应该把注意力集中在三个字段上：bi、bo和wa。bi字段告诉我们读取的块数；1,000相当于1 Mbps。bo字段是关于块的输出。它告诉我们写到磁盘上的数据量。在某种程度上，bi和bo是原始吞吐量。我不会认为一个数字是有害的。如果一个问题的wa值很高呢？bi和bo字段的低值，加上一个高的wa值，告诉我们一个潜在的磁盘瓶颈，这很可能与你系统上发生的大量随机I/O有关。wa值越高，你的查询速度就越慢，因为你在等待磁盘的响应。

良好的原始吞吐量是一件好事，但有时，它也可以指出一个问题。如果在线交易处理（OLTP）系统需要高吞吐量，它可以告诉你没有足够的RAM来缓存东西，或者索引丢失，PostgreSQL不得不读取太多的数据。请记住，事物是相互联系的，不应该把数据看作是孤立的。

7.了解值得注意的错误情况

在通过基本指南找出您将在数据库中遇到的最常见问题之后，接下来的部分将讨论PostgreSQL世界中发生的一些最常见的错误场景。

7.1 面对堵塞腐败

PostgreSQL有一个叫做提交日志的东西（现在叫做pg_xact；它的正式名称是pg_clog）。它跟踪系统中每个事务的状态，帮助PostgreSQL确定是否可以看到某一行。一般来说，一个事务可以处于四种状态。

```
#define TRANSACTION_STATUS_IN_PROGRESS 0x00
#define TRANSACTION_STATUS_COMMITTED 0x01
#define TRANSACTION_STATUS_ABORTED 0x02
#define TRANSACTION_STATUS_SUB_COMMITTED 0x03
```

在PostgreSQL数据库实例（pg_xact）中，clog有一个单独的目录。

在过去，人们曾报告过一种叫做clog损坏的东西，它可能是由有问题的磁盘或PostgreSQL中的bug引起的，这些年已经被修复了。一个损坏的提交日志是一个相当讨厌的东西，因为我们所有的数据都在那里，但PostgreSQL不知道事情是否仍然有效。这方面的损坏无异于一场彻底的灾难。

管理员是如何发现提交日志被破坏的呢？下面是我们通常看到的情况

```
ERROR: could not access status of transaction 118831
```

如果PostgreSQL不能访问一个事务的状态，就会出现问题。主要的问题是--如何才能解决这个问题？直截了当地告诉你，没有办法真正解决这个问题--我们只能尝试尽可能多地拯救数据。

正如我们已经说过的，提交日志为每个事务保留2位。这意味着我们每个字节有四个事务，每个块剩下32,768个事务。一旦我们弄清楚它是哪个块，我们就可以伪造事务日志：

```
dd if=/dev/zero of=<data directory location>/pg_clog/0001 bs=256K count=1
```

我们可以用dd来伪造事务日志，并将提交状态设置为所需的值。核心问题其实是--应该使用哪种事务状态？答案是，任何状态其实都是错的，因为我们真的不知道这些事务是如何结束的。

然而，通常情况下，为了减少数据损失，将它们设置为已提交是一个好主意。这真的取决于我们的工作量和我们的数据，以决定什么是较少的破坏性

当我们不得不使用这种技术时，我们应该在必要时尽可能少地伪造堵塞。记住，我们基本上是在伪造提交状态，这对数据库引擎来说不是一件好事。

一旦我们完成了伪造的堵塞，我们应该以最快的速度创建一个备份，并从头开始重新创建数据库实例。我们正在使用的系统不再是非常值得信赖的，所以我们应该尝试尽可能快地提取数据。请记住这一点：我们即将提取的数据可能是矛盾的和错误的，所以我们将确保对我们能够从数据库服务器中抢救出来的任何东西进行一些质量检查。

7.2 了解检查点消息

检查点对于数据完整性和性能至关重要。检查点相距越远，性能通常越好。在PostgreSQL中，默认配置通常相当保守，因此检查点相对较快。如果同时在数据库核心中更改大量数据，则PostgreSQL可能会告诉我们它认为检查点过于频繁。日志文件将显示以下条目：

```
LOG: checkpoints are occurring too frequently (2 seconds apart)
LOG: checkpoints are occurring too frequently (3 seconds apart)
```

在由于转储/恢复或其他一些大型操作导致的大量写入期间，PostgreSQL可能会注意到配置参数太低。一条消息被发送到LOG文件以准确地告诉我们

如果我们看到这种信息，出于性能的考虑，强烈建议我们通过大幅增加max_wal_size参数来增加检查点的距离（在旧版本中，该设置称为checkpoint_segments）。在最近的PostgreSQL版本中，默认配置已经比以前好很多了。然而，写入数据过于频繁的情况仍然容易发生

当我们看到关于检查点的信息时，有一件事我们必须牢记。过于频繁的检查点一点都不危险--它只是碰巧导致了糟糕的性能。写只是比原来慢了很多，但我们的数据没有危险。适当增加两个检查点之间的距离会使错误消失，同时也会加快我们的数据库实例的速度

7.3 管理损坏的数据页

PostgreSQL是一个非常稳定的数据库系统。它尽可能地保护数据，而且多年来已经证明了它的价值。然而，PostgreSQL依赖于坚实的硬件和正常工作的文件系统。如果存储系统坏了，PostgreSQL也会坏--除了增加复制以使事情更安全之外，我们对此没有什么办法。

偶尔会发生文件系统或磁盘故障的情况。然而，在许多情况下，整个系统不会出问题；只是有几个区块因某种原因而损坏。最近，我们已经看到这种情况发生在虚拟环境中。一些虚拟机默认不刷新磁盘，这意味着PostgreSQL不能依赖写入磁盘的东西。这种行为会导致难以预测的随机问题。

当一个区块不能再被读取时，你可能会遇到一个错误信息，如下面这样。

```
"could not read block %u in file "%s": %m"
```

你将要运行的查询会出错并停止工作。幸运的是，PostgreSQL有一个处理这些事情的方法。

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

zero_damaged_pages变量是一个配置变量，它允许我们处理破碎的页面。PostgreSQL不会抛出一个错误，而是将该区块简单地填充为零。

请注意，这肯定会导致数据丢失。但请记住，无论如何，这些数据之前就已经损坏或丢失了，所以这只是处理我们的存储系统中发生的坏事所造成的一种方法。

我建议大家小心处理zero_damaged_pages这个变量--当你调用它时要注意你在做什么。

7.4 粗心的连接管理

在PostgreSQL中，每个数据库连接都是一个独立的进程。所有这些进程都使用共享内存进行同步（技术上讲，在大多数情况下，它是映射的内存，但对于这个例子，这没有什么区别）。这个共享内存包含了I/O缓存、活动的数据库连接列表、锁和其他使系统正常运行的重要东西。

当一个连接被关闭时，它将从共享内存中删除所有相关的条目，并使系统处于正常的状态。然而，当一个数据库连接由于某种原因简单地崩溃时，会发生什么呢？

Postmaster（主进程）将检测到其中一个子进程丢失。然后，所有其他的连接将被终止，一个前滚进程将被初始化。为什么必须这样做呢？当一个进程崩溃时，很可能发生共享内存区被该进程编辑的情况。换句话说，一个崩溃的进程可能会使共享内存处于损坏的状态。因此，postmaster会做出反应，在损坏在系统中蔓延之前将所有人踢出去。所有的内存都被清理了，每个人都必须重新连接

从终端用户的角度来看，这感觉就像PostgreSQL崩溃并重新启动了，但事实并非如此。由于一个进程不能对自己的崩溃（分段故障）或其他一些信号做出反应，为了保护你的数据，清理一切是绝对必要的。

如果你在一个数据库连接上使用kill -9命令，也会发生同样的情况。该连接不能捕捉信号（根据定义，-9不能被捕捉），因此，postmaster必须再次做出反应。

7.5 对抗表的膨胀

在处理PostgreSQL时，表的膨胀是最重要的问题之一。当我们面临糟糕的性能时，弄清楚是否有对象需要的空间比它们应该有的多得多，总是一个好主意。

我们怎样才能弄清表的膨胀发生在哪呢？请看pg_stat_user_tables视图。

```
test=# \d pg_stat_user_tables
  View "pg_catalog.pg_stat_user_tables"
 Column | Type | Modifiers
+-----+-----+
 relid | oid |
 schemaname | name |
 relname | name |
 ...
 n_live_tup | bigint |
 n_dead_tup | bigint |
```

n_live_tup和n_dead_tup字段让我们对正在发生的事情有一个印象，我们也可以使用pgstattuple

如果出现了严重的表膨胀，我们可以做什么呢？第一个选择是运行VACUUM FULL命令。问题是，VACUUM FULL子句需要一个表锁。在一个大表上，这可能是一个真正的问题，因为在表被重写的时候，用户不能写到该表上

如果你至少使用PostgreSQL 9.6，你可以使用一个叫做pg_squeeze的工具。它在幕后组织一个表，而不会阻塞（<https://www.cybertec-postgresql.com/en/products/pg.squeeze/>）。如果你要重新组织一个非常大的表，这特别有用

8.总结

在这一章中，我们已经学会了如何系统地接近数据库系统，并检测人们在使用PostgreSQL时所面临的最常见的问题。我们了解了一些重要的系统表，以及其他一些可以决定我们是成功还是失败的重要因素。

在本书的最后一章，我们将把注意力放在迁移到PostgreSQL上。如果你正在使用Oracle或其他一些数据库系统，你可能想看看PostgreSQL。在第13章，迁移到PostgreSQL，我们将讨论与此有关的一切。

9.问题

- 为什么数据库不能自我管理？
- PostgreSQL是否经常遇到损坏？
- PostgreSQL是否需要经常照顾？

迁移到 PostgreSQL

1. 将SQL语句迁移到PostgreSQL

- 1.1 使用横向连接
 - 1.1.1 支持横向连接
- 1.2 使用分组集
 - 1.2.1 支持分组集
- 1.3 使用 WITH 子句——公用表表达式
 - 1.3.1 支持 WITH 子句
 - 1.3.2 使用 WITH RECURSIVE 子句
 - 1.3.3 支持 WITH RECURSIVE 子句
- 1.4 使用 FILTER 子句
 - 1.4.1 支持 FILTER 子句
- 1.5 使用窗口函数
 - 1.5.1 支持窗口和分析
- 1.6 使用有序集——WITHIN GROUP 子句
 - 1.6.1 支持 WITHIN GROUP 子句
- 1.7 使用 TABLESAMPLE 子句
 - 1.7.1 支持 TABLESAMPLE 子句
- 1.8 使用限制/偏移
 - 1.8.1 支持 FETCH FIRST 子句
- 1.9 使用 OFFSET 子句
 - 1.9.1 支持 OFFSET 子句
- 1.10 使用时态表
 - 1.10.1 支持时态表
- 1.11 时间序列中的匹配模式

2.从Oracle迁移到PostgreSQL

- 2.1 使用 oracle_fdw 扩展移动数据
- 2.2 使用 ora_migrator 进行快速迁移
- 2.3 CYBERTEC Migrator——“大男孩”的迁移
- 2.4 使用 Ora2Pg 从 Oracle 迁移
- 2.5 常见的陷阱

3.处理 MySQL 和 MariaDB 中的数据

- 3.1 更改列定义
- 3.2 处理空值
- 3.3 期待问题
- 3.4 迁移数据和模式
 - 3.4.1 使用 pg_chameleon
 - 3.4.2 使用FDWs

4.总结

在第12章 "PostgreSQL的故障排除"中，我们学习了如何处理与PostgreSQL故障排除有关的最常见的一个问题。重要的是要有一个系统的方法来追踪问题，这正是这里所提供的内容。

本书的最后一章是关于从其他数据库转移到PostgreSQL。你们中的许多人可能还在忍受商业数据库许可证费用带来的痛苦。我想给你们所有的人一条出路，告诉你们如何将数据从专有系统转移到PostgreSQL。转移到PostgreSQL不仅从财务角度来看是有意义的，而且如果你正在寻找更高级的功能和更多的灵活性，它也是有意义的。PostgreSQL有很多东西可以提供，在写这篇文章的时候，每天都在增加新的功能。这同样适用于可用于迁移到PostgreSQL的工具的数量。事情正在变得越来越好，而且开发人员一直在发布更多更好的工具。

本章将涵盖以下主题：

- 将SQL语句迁移到PostgreSQL

- 从Oracle迁移到PostgreSQL

在本章结束时，你应该能够将一个基本的数据库从其他系统转移到PostgreSQL。

1. 将SQL语句迁移到PostgreSQL

当从一个数据库转移到PostgreSQL时，看一看并弄清楚哪个数据库引擎提供哪种功能是有意义的。移动数据和结构本身通常是相当容易的。然而，重写SQL可能就不容易了。因此，我决定包括一个部分，明确地关注SQL的各种高级功能以及它们在今天的数据库引擎中的可用性。

1.1 使用横向连接

在SQL中，横向连接基本上可以被看作是某种循环。这允许我们对一个连接进行参数化处理，并在LATERAL子句中多次执行所有的内容。下面是一个简单的例子。

```
test=# SELECT *
  FROM generate_series(1, 4) AS x,
       LATERAL (SELECT array_agg(y)
  FROM generate_series(1, x) AS y
 ) AS z;
x | array_agg
---+-----
 1 | {1}
 2 | {1,2}
 3 | {1,2,3}
 4 | {1,2,3,4}
(4 rows)
```

LATERAL子句将对x的每个实例进行调用。对终端用户来说，这基本上是某种循环。

1.1.1 支持横向连接

一个重要的SQL特性是横向连接。下面的列表显示了哪些引擎支持横向连接，哪些不支持。

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 从PostgreSQL 9.3开始支持
- SQLite: 不支持
- Db2 LUW: 从9.1版（2005年）开始支持
- Oracle: 从12c开始支持
- Microsoft SQL Server: 自2005年起支持，但使用不同的语法

1.2 使用分组集

如果我们想同时运行一个以上的聚合，分组集就非常有用。使用分组集可以加快聚合速度，因为我们不需要多次处理数据。

下面是一个例子。

```
test=# SELECT x % 2, array_agg(x)
  FROM generate_series(1, 4) AS x
 GROUP BY ROLLUP (1);
?column? | array_agg
-----+-
 0 | {2,4}
 1 | {1,3}
  | {2,4,1,3}
(3 rows)
```

PostgreSQL提供的不仅仅是ROLLUP子句。也支持CUBE和GROUPING SETS子句。

1.2.1 支持分组集

分组集对于在一个查询中生成不止一个聚合是必不可少的。下面的列表显示了哪些引擎支持分组集，哪些不支持。

- MariaDB: 从5.1开始只支持ROLLUP子句 (不完全支持)
- MySQL: 从5.0开始只支持ROLLUP子句 (不完全支持)
- PostgreSQL: 从PostgreSQL 9.5开始支持
- SQLite: 不支持
- Db2 LUW: 至少从1999年开始支持
- Oracle: 从9iR1开始支持 (2000年左右)
- Microsoft SQL Server: 自2008年起支持

1.3 使用 WITH 子句——公用表表达式

普通表表达式是一种在SQL语句中执行东西的好方法，但只有一次。PostgreSQL 将执行所有的 WITH 子句，并允许我们在整个查询中使用这些结果

这是一个简化的例子：

```
test=# WITH x AS (SELECT avg(id)
  FROM generate_series(1, 10) AS id)
  SELECT *, y - (SELECT avg FROM x) AS diff
  FROM generate_series(1, 10) AS y
 WHERE y > (SELECT avg FROM x);
y | diff
---+-
 6 | 0.5000000000000000
 7 | 1.5000000000000000
 8 | 2.5000000000000000
 9 | 3.5000000000000000
10 | 4.5000000000000000
(5 rows)
```

在这个例子中，WITH子句的公共表扩展 (CTE) 计算了由generate_series函数生成的时间序列的平均值。产生的x可以像表一样在查询中被使用。在我的例子中，x被使用了两次。

1.3.1 支持 WITH 子句

下面的列表显示了哪些引擎支持WITH子句，哪些不支持

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 从PostgreSQL 8.4开始支持
- SQLite: 从 3.8.3 开始支持
- Db2 LUW: 从8 (2000) 开始支持
- Oracle: 从9iR2开始支持
- Microsoft SQL Server: 自2005年起支持

请注意，在PostgreSQL中，CTE甚至可以支持写入（INSERT、UPDATE和DELETE条款）。据我所知，没有其他数据库能够真正做到这一点。

1.3.2 使用 WITH RECURSIVE 子句

WITH 子句有两种形式：

- 标准CTE，如上一节所示（使用WITH子句）
- 在SQL中运行递归的一种方法

上一节介绍了CTE的简单形式。在下一节中，我们将介绍递归版本。

1.3.3 支持 WITH RECURSIVE 子句

下面的列表显示了哪些引擎支持WITH RECURSIVE子句，哪些不支持。

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 从PostgreSQL 8.4开始支持
- SQLite: 从3.8.3开始支持
- Db2 LUW: 从7 (2000) 开始支持
- Oracle: 从11gR2开始支持（在Oracle中，通常使用CONNECT BY子句而不是WITH RECURSIVE子句更常见）。
- Microsoft SQL Server: 自2005年起支持

1.4 使用 FILTER 子句

在查看SQL标准本身时，你会注意到FILTER子句从SQL (2003) 开始就存在了。然而，实际上没有多少系统支持这个非常有用的语法元素。

下面是一个例子。

```
test=# SELECT count(*),
  count(*) FILTER (WHERE id < 5),
  count(*) FILTER (WHERE id > 2)
FROM generate_series(1, 10) AS id;
count | count | count
-----+-----+
 10  |   4  |   8
(1 row)
```

如果一个条件不能在正常的WHERE子句中使用，因为有其他的聚合需要数据，那么FILTER子句就很有用。在引入FILTER子句之前，可以通过更繁琐的语法形式来实现同样的目的。

```
SELECT sum(CASE WHEN .. THEN 1 ELSE 0 END) AS whatever FROM some_table;
```

1.4.1 支持 FILTER 子句

下面的列表显示了哪些引擎支持FILTER子句，哪些不支持。

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 从PostgreSQL 9.4开始支持
- SQLite: 不支持
- Db2 LUW: 不支持
- Oracle: 不支持
- Microsoft SQL Server: 不支持

1.5 使用窗口函数

本书中已经广泛地讨论了窗口和分析。因此，我们可以直接跳到SQL顺应性方面。

1.5.1 支持窗口和分析

下面的列表显示了哪些引擎支持Windows功能，哪些不支持。

- MariaDB: 在最新版本中支持
- MySQL: 在最新的版本中支持
- PostgreSQL: 从PostgreSQL 8.4开始支持
- SQLite: 不支持
- Db2 LUW: 从7版开始支持
- Oracle: 从8i版开始支持
- Microsoft SQL Server: 自2005年起支持

其他一些数据库，如Hive、Impala、Spark和NuoDB，也支持分析。

1.6 使用有序集——WITHIN GROUP 子句

有序集合对PostgreSQL来说是相当新的。有序集和普通聚合的区别在于，在有序集的情况下，数据被送入聚合的方式确实有区别。假设你想在你的数据中找到一个趋势--数据的顺序是相关的。

下面是一个计算中位数的简单例子。

```
test=# SELECT id % 2,
percentile_disc(0.5) WITHIN GROUP (ORDER BY id)
FROM generate_series(1, 123) AS id
GROUP BY 1;
?column? | percentile_disc
-----+-
0 | 62
1 | 61
(2 rows)
```

只有在有排序输入的情况下才能确定中位数。

1.6.1 支持 WITHIN GROUP 子句

下面的列表显示了哪些引擎支持Windows功能，哪些不支持。

- MariaDB: 不支持MySQL。不支持
- PostgreSQL: 从PostgreSQL 9.4开始支持
- SQLite: 不支持
- Db2 LUW: 支持的
- Oracle: 从9iR1版本开始支持的
- Microsoft SQL Server: 支持，但查询必须使用窗口功能进行重塑

1.7 使用 TABLESAMPLE 子句

长期以来，表采样一直是商业数据库供应商的真正优势。传统的数据库系统已经提供了很多年的采样。然而，这种垄断已经被打破。从PostgreSQL 9.5开始，我们也有了解决采样问题的方法。

下面是它的工作原理。

```
test=# CREATE TABLE t_test (id int);
CREATE TABLE
test=# INSERT INTO t_test
SELECT * FROM generate_series(1, 1000000);
INSERT 0 1000000
```

首先，创建一个包含100万行的表。然后，可以执行测试。

```
test=# SELECT count(*), avg(id) FROM t_test TABLESAMPLE BERNOULLI (1);
count | avg
-----+
 9802 | 502453.220873291165
(1 row)
test=# SELECT count(*), avg(id) FROM t_test TABLESAMPLE BERNOULLI (1);
count | avg
-----+
 10082 | 497514.321959928586
(1 row)
```

在这个例子中，同一个测试被执行了两次。每次都使用1%的随机样本。两次的平均值都很接近500万，所以从统计学的角度来看，结果是相当不错的。

1.7.1 支持 TABLESAMPLE 子句

下面的列表显示了哪些引擎支持TABLESAMPLE子句，哪些不支持。

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 从PostgreSQL 9.5开始支持
- SQLite: 不支持
- Db2 LUW: 从8.2版开始支持
- Oracle: 从8版开始支持
- Microsoft SQL Server: 自2005年起支持

1.8 使用限制/偏移

在SQL中限制一个结果是一个有点悲伤的故事。简而言之，每个数据库的做法都有些不同。尽管实际上有一个关于限制结果的SQL标准，但不是每个人都完全支持事情应该是这样的。限制数据的正确方法是实际使用以下语法。

```
test=# SELECT * FROM t_test FETCH FIRST 3 ROWS ONLY;
 id
-----
 1
 2
 3
(3 rows)
```

如果你以前从未见过这种语法，不要担心。你绝对不是一个人

1.8.1 支持 FETCH FIRST 子句

下面的列表显示了哪些引擎支持FETCH FIRST子句，哪些不支持。

- MariaDB：从5.1开始支持（通常，使用limit/offset）
- MySQL：从3.19.3开始支持（通常，使用limit/offset）
- PostgreSQL：从PostgreSQL 8.4开始支持（通常，使用limit/offset）
- SQLite：从2.1.0版开始支持
- Db2 LUW：从版本7开始支持
- Oracle：从版本12c开始支持（使用带row_num函数的子选择）
- Microsoft SQL Server：自2012年起支持（传统上，使用top-N）。

正如你所看到的，限制结果集是相当棘手的，当你把一个商业数据库移植到PostgreSQL时，你很可能会遇到一些专有的语法。

1.9 使用 OFFSET 子句

OFFSET子句类似于FETCH FIRST子句。它很容易使用，但它还没有被广泛采用。它不像FETCH FIRST子句那样糟糕，但它仍然倾向于成为一个问题。

1.9.1 支持 OFFSET 子句

下面的列表显示了哪些引擎支持OFFSET子句，哪些不支持。

- MariaDB：从5.1开始支持
- MySQL：自4.0.6起支持
- PostgreSQL：从PostgreSQL 6.5开始支持
- SQLite：自2.1.0版起支持
- Db2 LUW：从11.1版开始支持
- Oracle：从12c版开始支持
- Microsoft SQL Server：自2012年起支持

正如你所看到的，限制结果集是相当棘手的，当你把一个商业数据库移植到PostgreSQL时，很可能会遇到一些专有的语法。

1.10 使用时态表

一些数据库引擎提供了时间表来处理版本问题。不幸的是，在PostgreSQL中没有这种开箱即用的版本管理。所以，如果你是从Db2或Oracle迁移过来的，你需要做一些工作来将所需的功能移植到PostgreSQL上。基本上，在PostgreSQL方面改变一下代码并不是太难。然而，这确实需要一些人工干预--它不再是一个直接复制和粘贴的工作。

1.10.1 支持时态表

下面的列表显示了哪些引擎支持时态表，哪些不支持。

- MariaDB: 不支持
- MySQL: 不支持
- PostgreSQL: 不支持
- SQLite: 不支持
- Db2 LUW: 从10.1版本开始支持
- Oracle: 从12cR1版本开始支持
- Microsoft SQL Server: 自2016年起支持

1.11 时间序列中的匹配模式

在写这篇文章的时候，最新的SQL标准（SQL 2016）提供了一个功能，旨在寻找时间序列中的匹配。到目前为止，只有Oracle在其最新版本的产品中实现了这个功能。

在这一点上，没有其他数据库厂商跟随他们并增加类似的功能。如果你想在PostgreSQL中模拟这种最先进的技术，你必须与窗口函数和子选择一起工作。在Oracle中匹配时间序列模式是相当强大的；在PostgreSQL中实现此目的的查询类型不止一种。

2.从Oracle迁移到PostgreSQL

到目前为止，我们已经看到了如何在PostgreSQL中移植或使用最重要的高级SQL特性。鉴于这些介绍，现在是时候特别看一下迁移Oracle数据库系统了。

这些天，由于Oracle新的许可和商业政策，从Oracle迁移到PostgreSQL已经变得非常流行。在世界范围内，人们正在远离Oracle而采用PostgreSQL。

2.1 使用 oracle_fdw 扩展移动数据

我首选的将用户从Oracle转移到PostgreSQL的方法之一是Laurenz Albe的oracle_fdw扩展（https://git.hub.com/laurenz/oracle_fdw）。它是一个外来数据封装器（FDW），允许你将Oracle中的表表示为PostgreSQL中的表。oracle_fdw扩展是最复杂的FDW之一，它坚如磐石，有很好的文档，是免费和开源的。

安装oracle_fdw扩展需要你安装Oracle客户端库。幸运的是，已经有了可以开箱即用的RPM包（<http://www.oracle.com/technetwork/topics/linuxx86-64soft-092277.html>）。oracle_fdw扩展需要OCI驱动来与Oracle对话。除了现成的Oracle客户端驱动之外，还有一个oracle_fdw扩展本身的RPM包，它是由社区提供的。如果你没有使用基于RPM的系统，你可能不得不自己编译，这显然是可能的，但有点费力。

一旦软件被安装，就可以很容易地启用。

```
test=# CREATE EXTENSION oracle_fdw;
```

CREATE EXTENSION子句将扩展加载到你想要的数据库中。现在，可以创建一个服务器，并将用户映射到其在Oracle方面的对应方，如下所示。

```
test=# CREATE SERVER oraserver FOREIGN DATA WRAPPER oracle_fdw OPTIONS (dbserver
'//dbserver.example.com/ORADB');
test=# CREATE USER MAPPING FOR postgres SERVER oradb OPTIONS (user 'orauser',
password 'orapass');
```

现在，是时候获取一些数据了。我的首选方法是使用IMPORT FOREIGN SCHEMA子句来导入数据定义。IMPORT FOREIGN SCHEMA子句将为远程模式中的每个表创建一个外表，并将数据暴露在Oracle一侧，然后可以很容易地读取这些数据。

利用模式导入的最简单的方法是在PostgreSQL上创建单独的模式，这些模式只是容纳数据库的模式。然后，可以使用FDW轻松地将数据吸进PostgreSQL。本章的最后一节，迁移数据和模式，关于从MySQL的迁移，向你展示了一个如何用MySQL/MariaDB进行迁移的例子。请记住，IMPORT FOREIGN SCHEMA子句是SQL/MED标准的一部分，因此这个过程与MySQL/MariaDB是一样的。这适用于几乎所有支持IMPORT FOREIGN SCHEMA子句的FDW。

虽然oracle_fdw扩展为我们做了大部分的工作，但看看数据类型是如何被映射的仍然是有意义的。Oracle和PostgreSQL没有提供完全相同的数据类型，所以有些映射是由oracle_fdw扩展或者我们手动完成的。下表提供了一个关于类型映射的概述。左边一列显示的是Oracle的类型，右边一列显示的是潜在的PostgreSQL的对应关系。

Oracle type	Possible PostgreSQL types
CHAR	char, varchar, text
NCHAR	char, varchar, text
VARCHAR	char, varchar, text
VARCHAR2	char, varchar, text, json
NVARCHAR2	char, varchar, text
CLOB	char, varchar, text, json
LONG	char, varchar, text
RAW	uuid, bytea
BLOB	bytea
BFILE	bytea (read-only)
LONG RAW	bytea
NUMBER	numeric, float4, float8, char, varchar, text
NUMBER(n,m) with m<=0	numeric, float4, float8, int2, int4, int8, boolean, char, varchar, text
FLOAT	numeric, float4, float8, char, varchar, text
BINARY_FLOAT	numeric, float4, float8, char, varchar, text
BINARY_DOUBLE	numeric, float4, float8, char, varchar, text
DATE	date, timestamp, timestamptz, char, varchar, text
TIMESTAMP	date, timestamp, timestamptz, char, varchar, text
TIMESTAMP WITH TIME ZONE	date, timestamp, timestamptz, char, varchar, text
TIMESTAMP WITH LOCAL TIME ZONE	date, timestamp, timestamptz, char, varchar, text
INTERVAL YEAR TO MONTH	interval, char, varchar, text
INTERVAL DAY TO SECOND	interval, char, varchar, text
MDSYS.SDO_GEOGRAPHY	geometry (see "PostGIS support" below)

如果你想使用几何图形，请确保你的数据库服务器上已经安装了PostGIS。

oracle_fdw扩展的缺点是，它不能迁移开箱即用的存储过程。存储过程是一个有点特殊的东西，需要一些人工干预

2.2 使用 ora_migrator 进行快速迁移

虽然oracle_fdw是一个好的开始，但我们可以做得更好。ora_migrator (https://www.cybertec-postgresql.com/en/ora_migrator-moving-from-oracle-to-postgresql-even-faster/, https://github.com/cybertec-postgresql/ora_migrator) 是在oracle_fdw之上开发的，并以最有效的方式使用其所有功能。它是如何工作的？一旦你从我们的GitHub页面安装了ora_migrator，你就可以通过使用以下命令来启用这个扩展。

```
CREATE EXTENSION ora_migrator;
```

一旦该模块被安装，看一看ora_migrator在做什么是有意义的。让我们运行一个示例调用并检查输出。

```
SELECT oracle_migrate(server => 'oracle', only_schemas => '{LAURENZ,SOCIAL}');
NOTICE: Creating staging schemas "ora_stage" and "pgsql_stage" ...
NOTICE: Creating Oracle metadata views in schema "ora_stage" ...
NOTICE: Copying definitions to PostgreSQL staging schema "pgsql_stage" ...
NOTICE: Creating schemas ...
NOTICE: Creating sequences ...
NOTICE: Creating foreign tables ...
NOTICE: Migrating table laurenz.log ...
...
NOTICE: Migrating table social.email ...
NOTICE: Migrating table laurenz.numbers ...
NOTICE: Creating UNIQUE and PRIMARY KEY constraints ...
WARNING: Error creating primary key or unique constraint on table
laurenz.badstring
DETAIL: relation "laurenz.badstring" does not exist:
WARNING: Error creating primary key or unique constraint on table laurenz.hasnul
DETAIL: relation "laurenz.hasnul" does not exist:
NOTICE: Creating FOREIGN KEY constraints ...
NOTICE: Creating CHECK constraints ...
NOTICE: Creating indexes ...
NOTICE: Setting column default values ...
NOTICE: Dropping staging schemas ...
NOTICE: Migration completed with 4 errors.
oracle_migrate
-----
4
(1 row)
```

ora_migrator的工作方式如下。首先，它克隆了Oracle系统目录的一部分，并把这些数据放到PostgreSQL数据库的一个暂存模式中。然后，这些信息被转换，这样我们就可以在PostgreSQL上实际使用它来轻松地创建表、索引、视图等等。在这个阶段，我们进行数据类型转换等等。

最后，数据被复制过来，索引、约束和类似的东西被应用。

你刚才看到的是最简单的情况。oracle_migrate只是一个封装函数，因此你也可以自己一步一步地调用需要的各个步骤。文档显示了在哪个级别可以做什么，你将很容易地以一种简单的方式迁移对象。

与其他一些工具相比，ora_migrator并不试图去做那些实际上不可能正确完成的事情。ora_migrator不触及的最重要的组件是程序。基本上不可能完全自动地将程序从Oracle程序转换为PostgreSQL程序。因此，我们不应该尝试去转换它们。简而言之，迁移存储过程仍然是一个部分手工操作的过程。

ora_migrator正在稳步改进，并且从11.2版本开始，可以用于所有的Oracle版本。

2.3 CYBERTEC Migrator——“大男孩”的迁移

如果你正在寻找一个更全面的、具有24/7支持的商业解决方案，我们可以推荐你看一下CYBERTEC Migrator，它可以在我的网站 (<https://www.cybertec-postgresql.com/en/products/cybertec-migrator/>) 上找到。它带有内置的并行性、高级数据类型预测、零停机迁移、自动代码重写等功能。

在我们的测试中，我们已经看到了高达1.5GB/秒的传输速度，这是我目前所知的最快的实现。请查看我们的网站以了解更多。

2.4 使用 Ora2Pg 从 Oracle 迁移

早在FDW出现之前，人们就从Oracle迁移到了PostgreSQL。长期以来，高额的许可证费用一直困扰着用户，因此，多年来，迁移到PostgreSQL是一件很自然的事情。

替代oracle_fdw扩展的是一个叫做Ora2Pg的东西，它已经存在了很多年，可以从<https://github.com/darold/Ora2Pg>。Ora2Pg是用Perl编写的，有一个长期的新版本传统。

Ora2Pg所提供的功能是惊人的。

- 迁移整个数据库模式，包括表、视图、序列和索引（唯一、主、外键和检查约束）。
- 迁移用户和组的权限。
- 迁移分区的表。
- 能够导出预定义的函数、触发器、程序、包和包体。
- 迁移全部或部分数据（使用WHERE子句）。
- 完全支持Oracle BLOB对象作为PostgreSQL bytea。
- 能够将Oracle视图导出为PostgreSQL表。
- 能够导出Oracle用户定义的类型。
- PL/SQL代码到PL/pgSQL代码的基本自动转换。注意，完全自动转换所有东西是不可能的。然而，很多东西都可以自动转换。
- 能够将Oracle表导出为FDW表。
- 能够导出物化视图。
- 能够显示有关Oracle数据库内容的详细报告。
- 评估Oracle数据库的迁移过程的复杂性。
- 从文件中对PL/SQL代码进行迁移成本评估。
- 能够生成用于Pentaho数据集成器（Kettle）的XML文件。
- 能够将Oracle定位器和空间几何图形导出到PostGIS。
- 能够将数据库链接导出为Oracle FDWs。
- 能够将同义词作为视图导出。
- 能够将一个目录作为外部表或外部文件扩展的目录导出。
- 能够在多个PostgreSQL连接上调度一个SQL命令列表。
- 能够为测试目的在Oracle和PostgreSQL数据库之间执行一个差异功能

使用Ora2Pg乍看之下很难。然而，它实际上比看起来容易得多。其基本概念如下。

```
/usr/local/bin/Ora2Pg -c /some_path/new_Ora2Pg.conf
```

Ora2Pg需要一个配置文件来运行。这个配置文件包含了处理这个过程所需要的所有信息。基本上，默认的配置文件已经非常不错了，对于大多数迁移来说，它是一个很好的起点。在Ora2Pg语言中，一个迁移就是一个项目。

配置将驱动整个项目。当你运行它的时候，Ora2Pg会创建几个目录，里面有所有从Oracle提取的数据。

```
Ora2Pg --project_base /app/migration/ --init_project test_project
Creating project test_project.
/app/migration/test_project/
schema/
```

```
dblinks/
directories/
functions/
grants/
mviews/
packages/
partitions/
procedures/
sequences/
synonyms/
tables/
 tablespaces/
triggers/
types/
views/
sources/
functions/
mviews/
packages/
partitions/
procedures/
triggers/
types/
views/
data/
config/
reports/
Generating generic configuration file
Creating script export_schema.sh to automate all exports.
Creating script import_all.sh to automate all imports.
```

正如你所看到的，可以直接执行的脚本被生成。然后，生成的数据可以很好地导入PostgreSQL中。要准备好在这里和那里改变程序。并非所有的东西都能自动迁移，所以人工干预是必要的。

2.5 常见的陷阱

有一些非常基本的语法元素在Oracle中起作用，但在PostgreSQL中可能不起作用。本节列出了一些需要考虑的最重要的隐患。当然，这个列表绝不是完整的，但它应该为你指出正确的方向。

在Oracle中，你可能会遇到下面的语句。

```
DELETE mytable;
```

在PostgreSQL中，这个语句是错误的，因为PostgreSQL要求你在DELETE语句中使用FROM子句。好消息是，这种语句很容易解决。

接下来你可能会发现下面的情况。

```
SELECT sysdate FROM dual;
```

PostgreSQL既没有sysdate函数，也没有双重函数。双重函数部分很容易解决，因为你可以简单地创建一个返回一行的VIEW函数。在Oracle中，双函数的工作原理如下。

```
SQL> desc dual
Name Null? Type
-----
DUMMY VARCHAR2(1)
SQL> select * from dual;
D
-
X
```

在PostgreSQL中，同样可以通过创建以下VIEW函数来实现。

```
CREATE VIEW dual AS SELECT 'X' AS dummy;
```

sysdate函数也很容易解决。它可以用clock_timestamp()函数代替。

另一个常见的问题是缺乏数据类型，如VARCHAR2，以及缺乏只有Oracle支持的特殊函数。解决这些问题的一个好办法是安装orafce扩展，它提供了大部分通常需要的东西，包括最常用的函数。当然，查看<https://github.com/orafce/orafce>以了解更多关于orafce扩展的信息是有意义的。它已经存在了很多年，是一个坚实的软件。

最近的一项研究表明，如果有orafce扩展，orafce扩展有助于确保73%的Oracle SQL可以在PostgreSQL上执行而无需修改（由NTT完成）。最常见的隐患之一是Oracle处理外层连接的方式。

请看下面的例子

```
SELECT employee_id, manager_id
  FROM employees
 WHERE employees.manager_id(+) = employees.employee_id;
```

这种语法不是由PostgreSQL提供的，将来也不会有。因此，这个连接必须被改写成一个适当的外连接。

在本章中，你已经学到了一些关于如何从Oracle等数据库迁移到PostgreSQL的宝贵经验。将MySQL和MariaDB数据库系统迁移到PostgreSQL是相当容易的。其原因是，Oracle可能很昂贵，而且时常有点麻烦。这同样适用于Informix。然而，Informix和Oracle都有一个重要的共同点：CHECK约束被正确地兑现，数据类型被正确地处理。一般来说，我们可以有把握地认为，这些商业系统中的数据基本上是正确的，没有违反数据完整性和常识的最基本规则。

我们的下一个候选人则不同。你所知道的关于商业数据库的许多事情在MySQL中并不正确。术语NOT NULL对MySQL没有什么意义（除非你明确使用严格模式）。在Oracle、Informix、Db2和我所知道的所有其他系统中，NOT NULL是一条在所有情况下都要遵守的法律。MySQL在默认情况下并不重视这些约束。（虽然，公平地说，这在最近的版本中已被改变。严格模式直到最近才被默认打开。然而，许多旧的数据库仍然使用旧的默认设置）。

在迁移的情况下，这引起了一些问题。你打算如何处理那些在技术上有问题的数据呢？如果你的NOT NULL列突然显示出无数的NULL条目，你打算如何处理？MySQL并不只是在NOT NULL列中插入NULL值。它将根据数据类型插入一个空字符串或0，所以事情可能变得非常糟糕。

3.处理 MySQL 和 MariaDB 中的数据

你可能可以想象，而且你可能已经注意到，当涉及到数据库时，我远非毫无偏见。然而，我并不想把这变成对MySQL/MariaDB的盲目抨击。我们的真正目标是看看为什么MySQL和MariaDB从长远来看会是如此痛苦。我有偏见是有原因的，我真的想指出为什么是这样。

我们将要看到的所有事情都是非常可怕的，对整个迁移过程有严重的影响。我已经指出，MySQL有些特殊，本节将试图证明我的观点。

同样，下面的例子假设我们使用的是没有开启严格模式的MySQL/MariaDB版本，本章最初写的时候就是这样（截至PostgreSQL 9.6）。从PostgreSQL 10.0开始，严格模式已经开启，所以我们在这里要读的大部分内容只适用于旧版本的MySQL/MariaDB。

让我们从创建一个简单的表开始。

```
MariaDB [test]> CREATE TABLE data (
    id integer NOT NULL,
    data numeric(4, 2)
);
Query OK, 0 rows affected (0.02 sec)

MariaDB [test]> INSERT INTO data VALUES (1, 1234.5678);
Query OK, 1 row affected, 1 warning (0.01 sec)
```

到目前为止，这里没有什么特别之处。我们已经创建了一个由两列组成的表。第一列被明确地标记为NOT NULL。第二列应该包含一个数字值，它被限制在四位数。最后，我们添加了一个简单的行。你能看到一个即将爆炸的潜在地雷吗？很可能没有。然而，检查一下下面的列表

```
MariaDB [test]> SELECT * FROM data;
+----+-----+
| id | data |
+----+-----+
| 1  | 99.99 |
+----+-----+
1 row in set (0.00 sec)
```

如果我没记错的话，我们添加了一个四位数的数字，这本来就不应该起作用。然而，MariaDB却简单地改变了我的数据。当然，已经发出了警告，但这是不应该发生的，因为表的内容并不反映我们实际插入的内容。

让我们尝试在PostgreSQL中做同样的事情。

```
test=# CREATE TABLE data
(
    id integer NOT NULL,
    data numeric(4, 2)
);
CREATE TABLE
test=# INSERT INTO data VALUES (1, 1234.5678);
ERROR: numeric field overflow
DETAIL: A field with precision 4, scale 2 must round to an absolute value less
than 10^2.
```

表被创建了，就像以前一样，但与MariaDB/MySQL形成鲜明对比的是，PostgreSQL会出错，因为我们试图向表中插入一个明显不允许的值。如果数据库引擎不关心，那么明确定义我们想要的东西有什么意义呢？假设你中了彩票--你可能刚刚失去了几百万，因为系统已经决定了什么对你有利。

我一生都在与商业数据库作斗争，但我从未在任何昂贵的商业系统（Oracle、Db2、Microsoft SQL Server等）中看到过类似的事情。他们可能有自己的问题，但一般来说，数据就很好。

3.1 更改列定义

让我们看看如果要修改表定义会发生什么：

```
MariaDB [test]> ALTER TABLE data MODIFY data numeric(3, 2);
Query OK, 1 row affected, 1 warning (0.06 sec)
Records: 1 Duplicates: 0 Warnings: 1
```

你应该在这里看到一个问题：

```
MariaDB [test]> SELECT * FROM data;
+----+-----+
| id | data |
+----+-----+
| 1  | 9.99 |
+----+-----+
1 row in set (0.00 sec)
```

正如你所看到的，这些数据又被修改了。它一开始就不应该在那里，而且又被重新修改了一遍。记住，你可能又损失了钱，或者其他一些好的资产，因为MySQL试图变得很聪明。

这就是PostgreSQL中发生的情况。

```
test=# INSERT INTO data VALUES (1, 34.5678);
INSERT 0 1
test=# SELECT * FROM data;
 id | data
----+-----
 1  | 34.57
(1 row)
```

现在，让我们更改列定义：

```
test=# ALTER TABLE data ALTER COLUMN data
      TYPE numeric(3, 2);
ERROR: numeric field overflow
DETAIL: A field with precision 3, scale 2 must round to
an absolute value less than 10^1.
```

同样，PostgreSQL会出错，它不允许我们对我们的数据做讨厌的事情。在任何重要的数据库中，预计也会发生同样的情况。这个规则很简单。PostgreSQL和其他系统不会允许我们破坏我们的数据。

然而，PostgreSQL允许你做一件事。

```
test=# ALTER TABLE data
      ALTER COLUMN data
      TYPE numeric(3, 2)
      USING (data / 10);
ALTER TABLE
```

我们可以明确地告诉系统该如何行动。在这个例子中，我们明确地告诉PostgreSQL将该列的内容除以10。开发人员可以明确地提供应用于数据的规则。PostgreSQL不会试图变得聪明，这是有原因的。

```
test=# SELECT * FROM data;
+---+
| id | data |
+---+
| 1  | 3.46 |
+---+
(1 row)
```

数据完全符合预期。

3.2 处理空值

我们不想把这变成一个为什么MariaDB不好的章节，但我想在这里补充一个最后的例子，我认为这个例子是非常重要的。

```
MariaDB [test]> UPDATE data SET id = NULL WHERE id = 1;
Query OK, 1 row affected, 1 warning (0.01 sec)
Rows matched: 1 Changed: 1 Warnings: 1
```

id列被显式标记为NOT NULL：

```
MariaDB [test]> SELECT * FROM data;
+---+
| id | data |
+---+
| 0  | 9.99 |
+---+
1 row in set (0.00 sec)
```

很明显，MySQL和MariaDB认为空和零是一回事。让我试着用一个简单的比喻来解释这里的问题：如果你知道你的钱包是空的，这和不知道你有多少钱是不同的。在我写这几行字的时候，我不知道我带了多少钱（空=未知），但我100%肯定它比零要多得多（我很肯定地知道在从机场回家的路上有足够的钱给我心爱的汽车加油，如果你的口袋里什么都没有，这是很难做到的）。

这里有一些更可怕的消息。

```
MariaDB [test]> DESCRIBE data;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| id   | int(11) | NO  |  | NULL   |  |
| data | decimal(3,2) | YES |  | NULL   |  |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

MariaDB确实记得这个列应该是NOT NULL的；但是，它只是再次修改了你的数据。

3.3 期待问题

主要的问题是，我们可能在向PostgreSQL移动数据时遇到麻烦。试想一下，你想移动一些数据，而在PostgreSQL那边有一个NOT NULL的约束。我们知道，MySQL并不关心。

```
MariaDB [test]> SELECT
    CAST('2014-02-99 10:00:00' AS datetime) AS x,
    CAST('2014-02-09 10:00:00' AS datetime) AS y;
+-----+-----+
| x | y |
+-----+-----+
| NULL | 2014-02-09 10:00:00 |
+-----+
1 row in set, 1 warning (0.00 sec)
```

PostgreSQL肯定会拒绝2月99日（有充分的理由），但如果你明确禁止它，它也可能不接受NULL值（有充分的理由）。在这种情况下，你必须做的是以某种方式修复数据，以确保它遵守你的数据模型的规则，这些规则的存在是有原因的。你不应该对此掉以轻心，因为你可能不得不改变那些实际上首先是错误的数据。

3.4 迁移数据和模式

现在我已经解释了为什么迁移到PostgreSQL是一个好主意，并概述了一些最重要的问题，现在是我解释我们最终摆脱MySQL/MariaDB的一些可能选项的时候了。

3.4.1 使用 pg_chameleon

从MySQL/MariaDB迁移到PostgreSQL的一个方法是使用Federico Campoli的工具pg_chameleon，它可以从GitHub (https://github.com/the4thdoctor/pg_chameleon) 免费下载。它被明确地设计为将数据复制到PostgreSQL，并为我们做了很多工作，如转换模式。基本上，该工具执行了以下四个步骤。

1. pg_chameleon工具从MySQL读取模式和数据，并在PostgreSQL中创建一个模式。
2. 它在PostgreSQL中存储MySQL的主连接信息。
3. 它在PostgreSQL中创建主键和索引。
4. 它从MySQL/MariaDB复制到PostgreSQL。

pg_chameleon工具提供对DDL的基本支持，如CREATE、DROP、ALTER TABLE和DROP PRIMARY KEY。然而，由于MySQL/MariaDB的特性，它并不支持所有的DDL。相反，它涵盖了最重要的功能。

然而，pg_chameleon还有更多的功能。我已经广泛地指出，数据并不总是它应该是或被期望是的样子。pg_chameleon处理这个问题的方法是丢弃垃圾数据并将其存储在一个叫做sch_chameleon.t_discarded_rows的表中。当然，这并不是一个完美的解决方案，但考虑到相当低质量的输入，这是我想到的唯一合理的解决方案。我们的想法是让开发人员决定如何处理所有被破坏的行。pg_chameleon真的没有办法决定如何处理被别人破坏的东西。

最近，已经进行了大量的开发工作，并在该工具中进行了大量的工作。因此，强烈建议你查看GitHub页面并阅读所有的文档。在写这本书的时候，功能和错误的修复正在增加。鉴于本章的范围有限，这里不可能全面覆盖。

存储过程、触发器等需要特殊处理，只能手动处理。pg_chameleon工具不能自动处理这些东西。

3.4.2 使用FDWs

如果我们想从MySQL/MariaDB转移到PostgreSQL，有不止一种方法可以成功。使用FDWs是pg_chameleon的一种替代方法，它提供了一种快速获取模式以及数据的方法，并将其导入PostgreSQL中。连接MySQL和PostgreSQL的能力已经存在了相当长的时间，因此FDWs绝对是一个可以利用的领域，对你有利。

基本上，mysql_fdw扩展的工作方式就像外面的其他FDW一样。与其他不太知名的FDW相比，mysql_fdw扩展实际上相当强大，提供了以下功能。

- 写入MySQL/MariaDB
- 连接池
- WHERE子句下推（这意味着应用于表的过滤器实际上可以远程端执行，以获得更好的性能）
- 列下推（只从远程端获取需要的列；旧版本用于获取所有列，这导致更多的网络流量）
- 远程端的准备语句

使用mysql_fdw扩展的方法是利用IMPORT FOREIGN SCHEMA语句，它允许你将数据转移到PostgreSQL上。幸运的是，在Unix系统上这是相当容易做到的。让我们来看看详细的步骤。

1.我们要做的第一件事是从GitHub上下载代码。

```
git clone https://github.com/EnterpriseDB/mysql_fdw.git
```

2.然后，运行以下命令来编译FDW。注意，在你的系统上，路径可能有所不同。在本章中，我假设MySQL和PostgreSQL都在/usr/local目录下，在你的系统中可能不是这样。

```
$ export PATH=/usr/local/pgsql/bin/:$PATH  
$ export PATH=/usr/local/mysql/bin/:$PATH  
$ make USE_PGXS=1  
$ make USE_PGXS=1 install
```

3.一旦代码被编译，FDW就可以被添加到我们的数据库中。

```
CREATE EXTENSION mysql_fdw;
```

4.下一步是创建我们要迁移的服务器。

```
CREATE SERVER migrate_me_server  
FOREIGN DATA WRAPPER mysql_fdw  
OPTIONS (host 'host.example.com', port '3306');
```

5.一旦服务器被创建，我们就可以创建所需的用户映射。

```
CREATE USER MAPPING FOR postgres  
SERVER migrate_me_server  
OPTIONS (username 'joe', password 'public');
```

6.最后，是时候进行真正的迁移了。要做的第一件事是导入模式。我建议先为链接表创建一个特殊的模式。

```
CREATE SCHEMA migration_schema;
```

7.当运行IMPORT FOREIGN SCHEMA语句时，我们可以使用这个模式作为目标模式，所有的数据库链接都将存储在这里。这样做的好处是，我们可以在迁移后方便地删除它。

8.一旦我们完成了IMPORT FOREIGN SCHEMA语句，我们就可以创建真正的表。最简单的方法是使用CREATE TABLE子句所提供的LIKE关键字。它允许我们复制一个表的结构并创建一个真正的、本地的PostgreSQL表。幸运的是，如果你要克隆的表只是一个FDW，这也是可行的。这里有一个例子。

```
CREATE TABLE t_customer
(LIKE migration_schema.t_customer);
```

9.然后，我们可以对数据进行处理。

```
INSERT INTO t_customer
SELECT * FROM migration_schema.t_customer
```

这实际上是我们可以纠正数据，消除大块的行，或者对数据做一些处理的地方。考虑到数据的低质量来源，在第一次移动数据后，应用约束条件等可能会很有用。这可能会让人不那么痛苦。

一旦数据被导入，我们就准备部署所有的约束、索引等等。在这一点上，你实际上会开始看到一些令人讨厌的惊喜，因为正如我之前所说，你不能指望数据坚如磐石。一般来说，在MySQL的情况下，迁移可能是相当困难的。

4.总结

在这一章中，我们了解了如何将SQL语句迁移到PostgreSQL中，并且学会了将一个基本的数据库从其他系统迁移到PostgreSQL中。迁移是一个重要的话题，每天都有越来越多的人在采用PostgreSQL。

PostgreSQL 12有很多新的功能，比如改进的内置分区，以及很多其他的功能。在未来，我们将看到PostgreSQL所有领域的更多发展，特别是那些允许用户扩展更多，运行查询更快。我们还没有看到未来会有什么变化。