日志文件和系统统计信息

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，数据库工作并不只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 | | |
 usename | 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（usename；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 | | |
 blks_read | bigint | | |
 blks_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什么？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祖。

> 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_blks*），你的索引的缓存行为（idx_blks*），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_blks_read | bigint | | |
 heap_blks_hit | bigint | | |
 idx_blks_read | bigint | | |
 idx_blks_hit | bigint | | |
 toast_blks_read | bigint | | |
 toast_blks_hit | bigint | | |
 tidx_blks_read | bigint | | |
 tidx_blks_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 | | |
 usename | 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过程的进度，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_blks_total | bigint | | |
 heap_blks_scanned | bigint | | |
 heap_blks_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，它可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_blks_hit | bigint | | |
shared_blks_read | bigint | | |
shared_blks_dirtied | bigint | | |
shared_blks_written | bigint | | |
local_blks_hit | bigint | | |
local_blks_read | bigint | | |
local_blks_dirtied | bigint | | |
local_blks_written | bigint | | |
temp_blks_read | bigint | | |
temp_blks_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
```

这会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）。