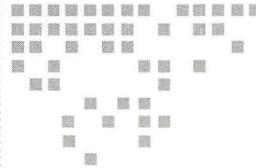




## 进 阶 篇

- 第 10 章 性能优化
- 第 11 章 基准测试与 pgbench
- 第 12 章 物理复制和逻辑复制
- 第 13 章 备份与恢复
- 第 14 章 高可用
- 第 15 章 版本升级
- 第 16 章 扩展模块
- 第 17 章 Oracle 数据库迁移 PostgreSQL 实践
- 第 18 章 PostGIS



*Chapter 10*

## 第 10 章

# 性能优化

为用户提供高性能的服务，是优秀的系统应该实现的目标，数据库的性能表现往往起到关键作用。在硬件层面，影响数据库性能的主要因素有 CPU、I/O、内存和网络；在软件层面则要复杂得多，操作系统配置、中间件配置、数据库参数配置、运行在数据库之上的查询和命令等，都对性能有或多或少的影响。同时，随着业务增长、数据量的变化，应用复杂度变更等种种因素的影响，数据库系统遇到瓶颈，运行在不健康状态的情况非常常见。本章将介绍一些关键的性能指标、判断性能瓶颈的方法，介绍查询计划的基础知识，以及一些常见的性能检测和系统监控工具，以及常见的性能瓶颈的解决思路，并着重介绍 PostgreSQL 丰富的索引类型和各种索引的使用场景，通过性能优化充分利用硬件资源，构建高效的 SQL 服务器应用。

规模稍大的应用系统，通常由若干子系统组成，例如一个应用由硬件、操作系统、PostgreSQL 数据库系统、业务系统组成，这些子系统一起工作，并且频繁交互，相互影响。在进行系统性能优化时，应当先着眼全局进行分析，再逐步深入到细节。PostgreSQL 数据库的 SQL 服务器应用通常分为 OLTP 和数据仓库，在当前日益复杂的数据应用环境中，这两种类型混合使用的情况越来越多，这也对数据库系统和数据库层面的优化提出了更高的要求。对于不同的应用类型，可能遇到的瓶颈也会不同，优化方法也大相径庭，以下就从服务器硬件、操作系统、数据库全局参数、查询性能这几个方面分别展开讨论。

## 10.1 服务器硬件

影响数据库性能的主要硬件因素有 CPU、磁盘、内存和网络。

最先到达瓶颈的，通常是磁盘的 I/O。在投入生产之前应该对磁盘的容量和吞吐量进行



估算，磁盘容量预估相对简单，但吞吐量受各种因素变化的影响，预估常常不够准确，所以要做好扩展的准备。固态存储现在已经非常成熟，而且价格已经比较便宜，目前在生产环境使用固态磁盘已经非常普遍，如目前使用广泛的 SATA SSD 和 PCIe SSD。与传统磁盘相比，SSD 有更好的随机读写性能，能够更好地支持并发，实现更大吞吐量，是现在数据库服务器首选的存储介质。但应该注意的是需要区分消费级 SSD 和企业级 SSD，如果在读写密集的生产环境使用廉价的消费级 SSD，不用多久，消费级 SSD 就会寿终正寝。使用外部存储设备加载到服务器也是比较常见的，例如 SAN（存储区域网络）和 NAS（网络接入存储）。使用 SAN 设备时通过块接口访问，对服务器来说就像访问本地磁盘一样；NAS 则是使用标准文件协议进行访问，例如 NFS 等。使用外部存储，还需要考虑网络通信对响应时间的影响。

CPU 也会经常成为性能瓶颈，数据库服务器执行的每一个查询都会给 CPU 施加一定压力。更高的 CPU 主频可以提供更快的运算速度，更多的核心数则能大大提高并发能力，充分利用 PostgreSQL 并发查询的能力。需要注意的是：有一些服务器在 BIOS 中可以设置 CPU 的性能模式，可能的模式有高性能模式、普通模式和节能模式，在数据库服务器中，不建议使用节能模式。这种模式会在系统比较空闲的时候对 CPU 主动降频，达到降温的目的，但对于数据库来说，则会产生性能波动，这是用户不希望看到的，所以硬件上架前就禁用节能模式，不同的设备请参考厂家提供的手册进行调整。

内存的使用对数据库系统非常重要。操作系统层、数据库层等各个层的缓存对高性能有很大辅助作用，较大的内存可以明显降低服务器的 I/O 压力，缓解 CPU 的 I/O 等待时间，对数据库性能起着关键的影响。但现在流行的服务器，内存配置一般都比较充裕，千兆和万兆网卡也几乎成为数据库服务器标配，内存与网络在大多数时候不会成为系统瓶颈，但仍然需要密切监控容量和指标趋势，在适当的时候进行扩容和升级。

## 10.2 操作系统优化

数据库是与操作系统结合非常紧密的系统应用，操作系统的参数配置会直接作用在数据库服务器。因此对于 DBA 来说，了解操作系统非常重要。下面我们介绍一些常用的性能监控和调整工具，并讨论一些常见的数据库专有服务器的操作系统方面的优化点，以及这些优化点相关参数的调整原则和方法。

### 10.2.1 常用 Linux 性能工具

Linux 操作系统提供了非常多的性能监控工具，可以全方位监控 CPU、内存、虚拟内存、磁盘 I/O、网络等各项指标，为问题排查提供了便利，无论是研发人员还是数据库管理员都应该熟练掌握这些工具和命令的用法。因为 Linux 相关命令和命令的变种很多，本节简单介绍一些常用的性能检测工具，例如 top、free、vmstat、iostat、mpstat、sar、pidstat 等。



除了 top 和 free 外，其他工具均位于 sysstat 包中。

在 CentOS 中安装 sysstat 包的命令如下所示：

```
[root@pghost1 ~]# yum install -y sysstat
```

### 1. top

top 命令是最常用的性能分析工具，它可以实时监控系统状态，输出系统整体资源占用状况以及各个进程的资源占用状况。在 top 命令运行过程中，还可使用一些交互命令刷新当前状态。一次 top 命令的输出如下所示：

```
top - 12:01:03 up 93 days, 23:30,  1 user,  load average: 1.08, 1.09, 1.08
Tasks: 1042 total,   3 running, 1039 sleeping,   0 stopped,   0 zombie
Cpu(s):  5.3%us,  1.6%sy,  0.0%ni, 92.5%id,  0.2%wa,  0.0%hi,  0.3%si,  0.0%st
Mem: 330604220k total, 323235920k used, 7368300k free, 248996k buffers
Swap: 67108860k total,          0k used, 67108860k free, 295053464k cached

      PID USER      PR NI VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
145138 postgres  20   0 37.7g 31g 31g S 26.2 10.1   5:38.22 postgres: pguser mydb
127.0.0.1(50382)
      58327 pgbounce  20   0 49368 8808 864 S 17.4  0.0  14725:14 /usr/bin/pgbounce
      -d -q /etc/pgbounce/pgbounce.ini
183682 postgres  20   0 37.6g 22g 21g R 17.3  7.0   1:49.69 postgres: pguser mydb
127.0.0.1(60026)
182679 postgres  20   0 37.7g 23g 22g S 15.3  7.4   2:11.07 postgres: pguser mydb
127.0.0.1(59112)
      58123 postgres  20   0 37.1g 36g 36g R 13.6 11.7  13623:27 postgres: startup
      process
164415 postgres  20   0 37.7g 26g 26g S 12.2  8.6   2:51.10 postgres: pguser mydb
127.0.0.1(42440)
      10674 root     20   0   0   0   0 S 11.7  0.0  22882:14 [shn_comp_wqa]
164421 postgres  20   0 37.7g 29g 28g S 11.6  9.2   3:55.88 postgres: pguser mydb
127.0.0.1(42452)
      57570 root     20   0   0   0   0 S  6.0  0.0   6386:38 [flush-252:0]
      73195 postgres  20   0 37.1g 3532 2280 S  6.0  0.0   6207:28 postgres: wal receiver
      process
      58145 postgres  20   0 37.1g 36g 36g S  3.6 11.6   4936:07 postgres: writer
      process
148192 postgres  20   0 37.7g 32g 31g S  3.4 10.3   5:57.01 postgres: pguser mydb
127.0.0.1(53174)
      10683 root     20   0   0   0   0 S  2.1  0.0   3233:57 [shn_handle_luna]
164413 postgres  20   0 37.7g 28g 27g S  2.1  9.0   3:31.48 postgres: pguser mydb
127.0.0.1(42436)
      10681 root     20   0   0   0   0 S  1.7  0.0   1579:57 [shn_gc_wqa]
      10675 root     20   0   0   0   0 S  1.1  0.0   886:09.06 [shn_wqa]
      73174 postgres  20   0 184m 5816 1036 S  1.0  0.0  745:02.12 postgres: stats
      collector process
      58144 postgres  20   0 37.2g 36g 36g S  0.3 11.6  812:08.97 postgres:
      checkpointer process
      ...
      ...
      ...
```



top命令的输出被一行空行分为两部分，空行以上的信息为服务器状态的整体统计信息，空行以下部分为各个进程的状态信息。

在本例中的统计信息区域如下所示：

```
top - 12:01:03 up 93 days, 23:30, 1 user, load average: 1.08, 1.09, 1.08
Tasks: 1042 total, 3 running, 1039 sleeping, 0 stopped, 0 zombie
Cpu(s): 5.3%us, 1.6%sy, 0.0%ni, 92.5%id, 0.2%wa, 0.0%hi, 0.3%si, 0.0%st
Mem: 330604220k total, 323235920k used, 7368300k free, 248996k buffers
Swap: 67108860k total, 0k used, 67108860k free, 295053464k cached
```

如果把这一部分输出翻译为可读语言，其内容如下所示：

```
top - 当前时间 12:01:03，系统已运行93天23小时30分没有重启，当前有1个用户登录操作系统，最近
1分钟、5分钟、15分钟的系统负载分别是：1.08, 1.09, 1.08
任务运行情况：当前一共有1042个进程，3个正在运行，1039个在睡眠，0个进程停止，0个僵尸进程
CPU：用户CPU占用5.3%，内核CPU占用1.6%，特定优先级的进程CPU占用0.0%，空闲CPU为92.5%，因为IO等
待的CPU占用0.2%，硬中断CPU占用0.0%，软中断CPU占用0.3%，虚拟机盗取占用0.0%
内存：共有330604220k，已使用323235920k，可用7368300k，buffers使用了248996k
虚拟内存：共有67108860k，已使用0k，可用67108860k，cache使用了295053464k
```

各个进程的状态信息区域的输出值所代表的含义是：

- ❑ PID：进程 id
- ❑ USER：进程所有者
- ❑ PR：进程优先级
- ❑ NI：进程优先级的修正值
- ❑ VIRT：进程使用的虚拟内存总量
- ❑ RES：进程使用的物理内存大小
- ❑ SHR：共享内存大小
- ❑ S：进程状态。D=不可中断的睡眠状态 R=运行 S=睡眠 T=跟踪 / 停止 Z=僵尸进程
- ❑ %CPU：上次更新到现在的 CPU 时间占用百分比
- ❑ %MEM：进程使用的物理内存百分比
- ❑ TIME+：进程使用的 CPU 时间总计，单位 1/100 秒
- ❑ COMMAND：进程运行的命令名

top命令默认情况下以 PID、USER、PR、NI、VIRT、RES、SHR、S、%CPU、%MEM、TIME+、COMMAND 从左到右的顺序输出这些列，通常情况下这些列的信息量已经足够丰富。进入交互页面可以选择添加删除不同的列，并可以对列进行排序。除了上述内容，top 命令功能丰富，可以阅读 man top 深入了解。

## 2. free

free命令显示当前系统的内存使用情况，如下所示：

```
[root@pghost1 ~]# free -g
              total        used         free       shared      buffers      cached

```

```
Mem: 315 306 9 36 0 278
-/+ buffers/cache: 27 287
Swap: 63 0 63
```

Mem 这一行的输出内容表示当前服务器的内存共有 315GB，已使用 306GB，可用 9GB，其中 total=used+free；共享内存为 36GB，buffers 使用了 0GB（这是由于输出是以 GB 为单位的，实际本例中是 244MB），cache 使用了 278GB；其中 buffers 和 cache 都是由操作系统为提高 I/O 性能而分配管理的，buffers 是将被写入到磁盘的缓冲区的数据，cache 是从磁盘读出到缓存的数据。-/+ buffers/cache 这一行前一个值是 used - buffers/cached 的值，是应用程序真正使用到的内存，在以上命令的输出中是 27GB；后一个值表示 free + buffers/cached 的值，表示理论上可以被使用的内存，在上述命令的输出中是 287GB。最后一行是总的 Swap 和可用的 Swap。

数据库在运行期间，会一直频繁地存取数据，对于操作系统而言也是频繁地存取数据。经过一段时间的运行，可能会发现 free 命令的输出结果中，可用内存越来越少，通常都是因为缓存。这是由于 Linux 为了提升 I/O 性能和减小磁盘压力，使用了 buffer cache 和 page cache，buffer cache 针对磁盘的写进行缓存，直接对磁盘进行操作的数据会缓存到 buffer cache，而文件系统中的数据则是交给 page cache 进行缓存，即使数据库任务运行结束，cache 也不会被主动释放。所以，是否使用到 Swap 可以作为判断内存是否够用的一个简单标准，只要没有使用到 Swap，就说明内存还够用，在数据库需要内存时，cache 可以很快被回收。

如果想把缓存释放出来，可以使用如下命令：

```
[root@pghost1 ~]# sync
[root@pghost1 ~]# echo 1 > /proc/sys/vm/drop_caches
```

需要注意，在生产环境释放缓存的命令要慎用，避免引起性能波动。

### 3. vmstat

vmstat 是 Linux 中的虚拟内存统计工具，用于监控操作系统的虚拟内存、进程、CPU 等的整体情况。vmstat 最常规的用法是：vmstat delay count，即每隔 delay 秒采样一次，共采样 count 次，如果省略 count，则一直按照 delay 时间进行采样，直到用户手动 CTRL+C 停止为止。举例如下：

```
[root@pghost1 ~]# vmstat 3 5
procs --memory-- --swap-- --io-- --system-- --cpu--
 r b swpd free buff cache si so bi bo in cs us sy id wa st
 3 0 7598968 243376 292787104 0 0 1354 984 0 0 4 2 94 1 0
 1 0 7491112 243388 292891072 0 0 27019 45264 45526 72922 4 2 94 0 0
 0 0 7434112 243408 292946016 0 0 11299 32468 45675 68756 4 1 94 0 0
 2 0 7348156 243468 293028032 0 0 19383 67697 48760 75728 4 2 94 0 0
 2 0 7289800 243480 293085504 0 0 12467 28441 46016 64854 4 1 95 0 0
```

vmstat 的输出第一行显示了系统自启动以来的平均值，从第二行开始显示现在正在发生的情况，每一列的含义如下：

- r：当前 CPU 队列中有几个进程在等待，持续为 1 说明有进程一直在等待，超过核心数说明压力过大；

- b：当前有多少个进程进入不可中断式睡眠状态；
- swpd：已经使用的交换分区的大小；
- free：当前的空闲内存；
- buff：已经使用的 buffer 的大小，特指 buffer cache（存在用来描述文件元数据的 cache）；
- cache：已经使用的 page cache 的大小，特指文件 page 的 cache；
- si/so：从磁盘交换到 Swap 分区和从 Swap 分区交换到磁盘的大小；
- bi/bo：从磁盘读出和写入到磁盘的大小，单位 blocks/s；
- in：每秒被中断的进程数；
- cs：每秒多少个 CPU 进程在进进出出。

#### 4. iostat

iostat 命令用于整个系统、适配器、tty 设备、磁盘和 CD-ROM 的输入 / 输出统计信息，但最常用的是用 iostat 来监控磁盘的输入输出，和 vmstat 一样，在命令的后面可以跟上 delay 和 count 参数，举例如下：

```
[root@pghost1 ~]# iostat -dx /dev/dfa 5 5
Linux 2.6.32-696.el6.x86_64 (pghost1)        01/25/2018      _x86_64_      (48 CPU)
Device:      rrqm/s   wrqm/s     r/s     w/s   rsec/s   wsec/s  avgrq-sz avgqu-
              sz   await r_await w_await svctm %util
dfa          0.00     0.00 3679.66 11761.85 129603.44 94094.80    14.49     0.17
  0.03     0.01     0.04   0.02  27.97
Device:      rrqm/s   wrqm/s     r/s     w/s   rsec/s   wsec/s  avgrq-sz avgqu-
              sz   await r_await w_await svctm %util
dfa          0.00     0.00  807.60  9619.20 33854.40 76953.60    10.63     3.89
  0.37     0.15     0.39   0.01  14.00
Device:      rrqm/s   wrqm/s     r/s     w/s   rsec/s   wsec/s  avgrq-sz avgqu-
              sz   await r_await w_await svctm %util
dfa          0.00     0.00  727.80 12904.40 29558.40 103235.20    9.74     7.39
  0.54     0.18     0.56   0.01  15.18
Device:      rrqm/s   wrqm/s     r/s     w/s   rsec/s   wsec/s  avgrq-sz avgqu-
              sz   await r_await w_await svctm %util
dfa          0.00     0.00 1682.00 13761.20 52283.20 110089.60   10.51    16.64
  1.07     0.15     1.18   0.01  20.76
Device:      rrqm/s   wrqm/s     r/s     w/s   rsec/s   wsec/s  avgrq-sz avgqu-
              sz   await r_await w_await svctm %util
dfa          0.00     0.00  609.80 23131.60 24737.60 185052.80    8.84    16.50
  0.69     0.24     0.70   0.01  21.62
```

以上输出的每列的含义是：

- rrqm/s, wrqm/s：每秒读写请求的合并数量（OS 会尽量读取和写入临近扇区）；
- r/s, w/s：每秒读写请求次数；

- rsec/s, wsec/s: 每秒读写请求的字节数;
- avgrq-sz: 每秒请求的队列大小;
- avgqu-sz: 每秒请求的队列长度;
- await: 从服务发起到返回信息共花费的平均服务时间;
- svctm: 该值不必关注;
- %util: 磁盘的利用率。

## 5. mpstat

mpstat 返回 CPU 的详细性能信息，举例说明：

```
[root@pghost1 ~]# mpstat 5 5
Linux 2.6.32-696.el6.x86_64 (pghost1)        01/25/2018      _x86_64_      (48 CPU)
02:54:30 PM  CPU  %usr   %nice   %sys %iowait   %irq   %soft   %steal   %guest   %idle
02:54:35 PM    all   3.88     0.00    1.16     0.33     0.00    0.22     0.00     0.00    94.41
02:54:40 PM    all   4.06     0.00    0.98     0.14     0.00    0.24     0.00     0.00    94.58
02:54:45 PM    all   4.06     0.00    0.98     0.13     0.00    0.23     0.00     0.00    94.60
02:54:50 PM    all   4.15     0.00    1.37     0.37     0.00    0.24     0.00     0.00    93.88
02:54:55 PM    all   6.15     0.00    1.16     0.15     0.00    0.24     0.00     0.00    92.30
Average:      all   4.46     0.00    1.13     0.22     0.00    0.23     0.00     0.00    93.95
```

在 mpstat 的最后一行会有一个运行期间的平均统计值，默认的 mpstat 会统计所有 CPU 的信息，如果只需要观察某一个 CPU，加上参数 -P n，n 为要观察的 core 的索引。例如观察 CPU 0 的统计信息，每 5 秒采样一次，共采样 3 次，命令如下所示：

```
[root@pghost1 ~]# mpstat -P 0 5 3
Linux 2.6.32-696.el6.x86_64 (pghost1)        01/25/2018      _x86_64_      (48 CPU)
02:57:16 PM  CPU  %usr   %nice   %sys %iowait   %irq   %soft   %steal   %guest   %idle
02:57:21 PM    0   17.54     0.00    2.02     0.81     0.00    0.40     0.00     0.00    79.23
02:57:26 PM    0   19.07     0.00    2.43     0.61     0.00    0.41     0.00     0.00    77.48
02:57:31 PM    0   18.96     0.00    3.39     1.00     0.00    0.60     0.00     0.00    76.05
Average:      0   18.52     0.00    2.62     0.81     0.00    0.47     0.00     0.00    77.58
```

以上输出的各列的含义如下：

- %usr: 用户花费的时间比例;
- %nice: 特定优先级进程的 CPU 时间百分比;
- %sys: 系统花费的时间比例;
- %iowait: I/O 等待;
- %irq: 硬中断花费的 CPU 时间;
- %soft: 软中断花费的 CPU 时间;
- %steal,%guest: 这两个参数与虚拟机相关，略;
- %idle: 空闲比率。

## 6. sar

sar 是性能统计非常重要的工具。sar 每隔一段时间进行一次统计，它的配置在 /etc/

cron.d/sysstat 中，默认为 10 分钟：

```
[root@pghost1 ~]# cat /etc/cron.d/sysstat
# Run system activity accounting tool every 10 minutes
*/10 * * * * root /usr/lib64/sa/sa1 1 1
```

可以调整统计信息的收集频率，在测试过程中可以将每 10 分钟修改为每 1 分钟用来提高统计的实时性。

sar 的结果是基于历史的，即从开机到最后一次收集数据时的统计信息，在输出时默认使用 AM/PM 显示时间，可以在执行 sar 前强制使用 LANG=C 以使用 24 小时时间表示法显示时间；

sar 统计的维度很多，下面举几个简单的例子。

### (1) 汇总 CPU 状况

汇总 CPU 状况的命令如下所示：

```
[root@pghost1 ~]# sar -q
12:00:01 AM    runq-sz   plist-sz   ldavg-1   ldavg-5   ldavg-15
12:10:01 AM        9       1306      1.17      1.17      1.19
12:20:01 AM        4       1307      1.21      1.19      1.19
...
...
...
10:20:01 AM        9       1297      1.02      1.09      1.13
10:30:01 AM        7       1300      0.95      0.89      0.99
10:40:01 AM        8       1298      1.48      1.27      1.10
...
...
...
```

以上输出的含义为：

- runq-sz：运行队列平均长度；
- plist-sz：进程列数；
- ldavg-1, ldavg-5, ldavg-15：每分、每 5 分钟、每 15 分钟的平均负载。

### (2) 汇总 I/O 状况

汇总 I/O 状况的命令如下所示：

```
[root@pghost1 ~]# sar -b
12:00:01 AM      tps      rtps      wtps   bread/s   bwrttn/s
12:10:01 AM  13995.40    947.73  13047.68  35553.77 104404.64
12:20:01 AM  14389.13   1162.41  13226.72  40502.67 105837.12
12:30:01 AM  16420.53   1107.85  15312.69  43851.18 122524.99
...
...
...
01:00:01 PM  15961.31   1444.19  14517.12  52564.87 116160.35
01:10:01 PM  15327.42   1201.94  14125.48  43332.46 113027.64
```

...

...

...

以上输出的含义为：

- tps, rtps, wtps: TPS 数；

- bread/s, bwrtn/s：每秒读写 block 的大小，需要注意：这里每个 block 的大小为 512 字节，不要与数据库中的 Block 混淆了。

### (3) 历史数据的汇总

sar 的历史数据保存在 /var/log/sa/ 目录，可以设置 sar 历史数据的保留天数，查看默认保存几天的历史数据，要修改保存天数可以编辑 /etc/sysconfig/systat 中的 HISTORY 值。当设置的保存天数超过 28 天，则会在 /var/log/sa/ 下建立月份目录。只查看某一天的数据指定一下具体的日期对应的历史数据文件即可，例如查看 15 号 22:00:00 到 23:00:00 的 CPU 性能统计数据，命令如下所示：

```
[root@pghost1 ~]# sar -q -f /var/log/sa/sa15 -s 22:00:00 -e 23:00:00
10:00:01 PM    runq-sz  plist-sz   ldavg-1    ldavg-5  ldavg-15
10:10:01 PM        5       1311      1.38      1.44      1.36
10:20:01 PM        2       1312      1.15      1.18      1.25
10:30:01 PM        5       1313      1.50      1.20      1.19
10:40:01 PM        4       1312      1.01      1.20      1.16
10:50:01 PM        6       1334      1.64      1.25      1.18
Average:          4       1316      1.34      1.25      1.23
```

sar 是性能统计信息的集大成者，是 Linux 上最为全面的性能分析工具之一，保存的历史数据可以借助 gnuplot 工具绘制性能指标图形，还可以使用 grafana 等图形前端进行性能指标的趋势图绘制，直观地观察性能趋势。

## 7. 其他性能工具和方法

nmon，像一个图形界面的 top 一样，可以动态地、漂亮地显示当前的 I/O、CPU、存储、网络的实时性能，并且可以将历史数据通过 OFFICE 宏输出为图表。

iostop，像 top 工具一样，但它是用来观察 I/O 状况的，可以方便地观察当前系统的 I/O 是否存在瓶颈以及在 I/O 出现瓶颈时观察是哪些进程造成的。通常还会配合 pidstat 命令一起来排查问题。

除了使用 iostop 这样专业的工具来定位 I/O 问题，还可以直接利用进程状态来找到相关的进程。

我们知道进程有如下几种状态：

- D 不可中断的睡眠状态。
- R 可执行状态。
- S 可中断的睡眠状态。
- T 暂停状态或跟踪状态。

- X dead 退出状态，进程即将被销毁。

- Z 退出状态，进程成为僵尸进程。

其中状态为 D 的进程一般就是由于等待 I/O 而造成所谓的“非中断睡眠”，我们可以从这点入手然后一步步地定位问题，如下所示：

```
[root@pghost1 ~]# for x in `seq 3`; do ps -eo state,pid,cmd | grep "^\D"; echo "--"; sleep 5; done;
```

或

```
[root@pghost1 ~]# while true; do date; ps auxf | awk '{if($8=="D") print $0;}'; sleep 1; done;
```

还可以用 pidstat -d 1 来查看进程的读写情况，用来诊断 I/O 问题。

### 10.2.2 Linux 系统的 I/O 调度算法

磁盘 I/O 通常会首先成为数据库服务器的瓶颈，因此我们先简单了解在 Linux 系统中的 I/O 调度算法，并根据不同的硬件配置，调整调度算法提高数据库服务器性能。对于数据库的读写操作，Linux 操作系统在收到数据库的请求时，Linux 内核并不是立即执行该请求，而是通过 I/O 调度算法，先尝试合并请求，再发送到块设备中。

通过如下命令查看当前系统支持的调度算法：

```
[root@pghost1 ~]# dmesg | grep -i scheduler
io scheduler noop registered
io scheduler anticipatory registered
io scheduler deadline registered
io scheduler cfq registered (default)
```

cfq 称为绝对公平调度算法，它为每个进程和线程单独创建一个队列来管理该进程的 I/O 请求，为这些进程和线程均匀分布 I/O 带宽，比较适合于通用服务器，是 Linux 系统中默认的 I/O 调度算法。

noop 称为电梯调度算法，它基于 FIFO 队列实现，所有 I/O 请求先进先出，适合 SSD。

deadline 称为绝对保障算法，它为读和写分别创建了 FIFO 队列，当内核收到请求时，先尝试合并，不能合并则尝试排序或放入队列中，并且尽量保证在请求达到最终期限时进行调度，避免有一些请求长时间不能得到处理，适合虚拟机所在宿主机器或 I/O 压力比较重的场景，例如数据库服务器。

可以通过以下命令查看磁盘 sda 的 I/O 调度算法：

```
[root@pghost1 ~]# cat /sys/block/sda/queue/scheduler
noop anticipatory deadline [cfq]
```

在输出结果中，被方括号括起来的值就是当前 sda 磁盘所使用的调度算法。

通过 shell 命令可以临时修改 I/O 调度算法：

```
[root@pghost1 ~]# echo noop > /sys/block/sda/queue/scheduler
[root@pghost1 ~]# cat /sys/block/sda/queue/scheduler
[noop] anticipatory deadline cfq
```

shell 命令修改的调度算法，在服务器重启后就会恢复到系统默认值，永久修改调度算法需要修改 /etc/grub.conf 文件。

### 10.2.3 预读参数调整

除了根据不同应用场景，配置磁盘的 I/O 调度方式之外，还可以通过调整 Linux 内核预读磁盘扇区参数进行 I/O 的优化。在内存中读取数据比从磁盘读取要快很多，增加 Linux 内核预读，对于大量顺序读取的操作，可以有效减少 I/O 的等待时间。如果应用场景中有大量的碎片小文件，过多的预读会造成资源的浪费。所以该值应该在实际环境多次测试。

通过如下命令查看磁盘预读扇区：

```
[root@pghost1 ~]# /sbin/blockdev --getra /dev/sda
256
```

默认为 256，在当前较新的硬件机器中，可以设置到 16384 或更大。通过以下命令设置磁盘预读扇区：

```
[root@pghost1 ~]# /sbin/blockdev --setra 16384 /dev/sda
```

或

```
[root@pghost1 ~]# echo 16384 /sys/block/sda/queue/read_ahead_kb
```

为防止重启失效，可以将配置写入 /etc/rc.local 文件，对多块磁盘设置该值，如下所示：

```
[root@pghost1 ~]# echo "/sbin/blockdev --setra 16384 /dev/dfa /dev/sda1" >> /etc/
rc.local
[root@pghost1 ~]# cat /etc/rc.local
#!/bin/sh
...
...
...
# Database optimisation
/sbin/blockdev --setra 16384 /dev/dfa /dev/sda1
```

### 10.2.4 内存的优化

#### 1. Swap

在内存方面，对数据库性能影响最恶劣的就是 Swap 了。当内存不足，操作系统会将虚拟内存写入磁盘进行内存交换，而数据库并不知道数据在磁盘中，这种情况下就会导致性能急剧下降，甚至造成生产故障。有些系统管理员会彻底禁用 Swap，但如果这样，一旦内存消耗完就会导致 OOM，数据库也会随之崩溃。

查看系统是否已经使用到了 Swap 最简单的方法就是 free 命令了，如下所示：

```
[postgres@pghost1 ~]$ free -g
              total        used        free      shared      buffers      cached
Mem:          378         34       344         30          0         30
-/+ buffers/cache:         3       375
Swap:         31          0         31
```

通过以上命令及其输出可以看到，Swap 共分配了 31GB，实际使用为 0，说明并没有使用到 Swap。

还可以使用 vmstat 之类的命令来查看，如下所示：

```
[postgres@pghost1 ~]$ vmstat 5 3
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     0 361127872 390616 31882912   0   0    2   11    0    0  0  0 100  0  0
0 0     0 361127840 390620 31882912   0   0    0   30  744  533  0  0 100  0  0
1 0     0 361127840 390620 31882912   0   0    0   22 1161  694  0  0 100  0  0
[postgres@pghost1 ~]$
```

在 vmstat 的输出结果中第三列 swpd 如果大于 0，则说明使用到了 Swap，在上述例子中并没有使用 Swap。

如果由于特殊原因，已经用到了 Swap，那么应该在有可用内存时释放已使用的 Swap，释放 Swap 的过程实际上是先禁用 Swap 后再启用 Swap 的过程。禁用 Swap 的命令是 swapoff，启用 Swap 的命令是 swapon，例如：

```
[root@pghost1 ~]# swapoff -a
[root@pghost1 ~]#
[root@pghost1 ~]# free | grep Swap
Swap:          0          0          0
[root@pghost1 ~]#
[root@pghost1 ~]# swapon -a
[root@pghost1 ~]#
[root@pghost1 ~]# free | grep Swap
Swap: 33554428          0 33554428
```

禁用 Swap 之后，可以看到可用的交换空间值为 0，在启用 Swap 之后，可以看到可用空间是预先划分的 Swap 分区的大小。

## 2. 透明大页

透明大页（Transparent HugePages）在运行时动态分配内存，而运行时的内存分配会有延誤，对于数据库管理系统来说并不友好，所以建议关闭透明大页。

查看透明大页的系统配置的命令如下所示：

```
[root@pghost1 ~]# cat /sys/kernel/mm/transparent_hugepage/enabled
[always] madvise never
```

在上述命令的输出中方括号包围的值就是当前值，关闭透明大页的方法如下所示：

```
[root@pghost1 ~]# echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

```
[root@pghost1 ~]# cat /sys/kernel/mm/transparent_hugepage/enabled
always madvise [never]
```

永久禁用透明大页可以通过编辑 /etc/rc.local，加入以下内容：

```
if test -f /sys/kernel/mm/transparent_hugepage/enabled; then
    echo never > /sys/kernel/mm/transparent_hugepage/enabled
fi
if test -f /sys/kernel/mm/transparent_hugepage/defrag; then
    echo never > /sys/kernel/mm/transparent_hugepage/defrag
fi
```

还可以通过修改 /etc/grub.conf，在 kernel 的行末加上 transparent\_hugepage=never 禁用透明大页，如下所示：

```
kernel /boot/vmlinuz-2.6.32-642.11.1.el6.x86_64 ro root=UUID=c429e8ae-c35d-4bc0-
a781-17bbb95a75cf nomodeset rd_NO_LUKS KEYBOARDTYPE=pc KEYTABLE=us LANG=en_
US.UTF-8 rd_NO_MD SYSFONT=latarcyrheb-sun16 crashkernel=auto rd_NO_LVM rd_NO_
DM rhgb quiet numa=off transparent_hugepage=never
```

### 3. NUMA

NUMA 架构会优先在请求线程所在的 CPU 的 local 内存上分配空间，如果 local 内存不足，优先淘汰 local 内存中无用的页面，这会导致每个 CPU 上的内存分配不均，虽然可以通过配置 NUMA 的轮询机制缓解，但对于数据库管理系统仍不太好，建议关闭 NUMA。

查看 NUMA 在操作系统中是否开启的命令如下所示：

```
[root@pghost1 ~]# numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46
node 0 size: 196514 MB
node 0 free: 186290 MB
node 1 cpus: 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47
node 1 size: 196608 MB
node 1 free: 192336 MB
node distances:
node     0      1
0:   10    21
1:    21    10
```

或使用 numastat 命令，如下所示：

```
[root@pghost1 ~]# numastat
              node0            node1
numa_hit        27207118        27526494
numa_miss         0                 0
numa_foreign       0                 0
interleave_hit     111148        111125
local_node        27205425        27405472
other_node         1693         121022
```

通过 numactl 的输出，目前可以看到两个 CPU 内存节点：available: 2 nodes (0-1)，并



且这两个节点所分配的内存大小是不一样的。

关闭 NUMA 最直接的方法是从服务器的 BIOS 中关闭，例如某品牌服务器的关闭方法是禁用内存配置中的 Node Interleaving。其手册中的说明如下：

Node Interleaving（节点交叉存取）指定是否支持非统一内存架构。如果此字段设为 Enabled（已启用），当安装的是对称内存配置时，支持内存交叉存取。如果此字段设为 Disabled（已禁用），系统支持 NUMA（非对称）内存配置。在默认情况下，该选项设为 Disabled（禁用）。

还可以通过编辑 /etc/grub.conf，在 kernel 的行末加上 numa=off 禁用 NUMA，如下所示：

```
kernel /boot/vmlinuz-2.6.32-642.11.1.el6.x86_64 ro root=UUID=c429e8ae-c35d-4bc0-a781-17bbb95a75cf nomodeset rd_NO_LUKS KEYBOARDTYPE=pc KEYTABLE=us LANG=en_US.UTF-8 rd_NO_MD SYSFONT=latarcyrheb-sun16 crashkernel=auto rd_NO_LVM rd_NO_DM rhgb quiet numa=off
```

关闭之后再查看 NUMA 的状态，如下所示：

```
[root@pghost1 ~]# numactl --hardware
available: 1 nodes (0)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
          28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
node 0 size: 393122 MB
node 0 free: 379902 MB
node distances:
node 0
 0: 10
[root@pghost1 ~]#
[root@pghost1 ~]# numastat
               node0
numa_hit           3613403
numa_miss           0
numa_foreign         0
interleave_hit     222419
local_node          3613403
other_node           0
```

关闭之后观察数据库的表现会发现性能有一定幅度提升，并且原本有一些小波动的地方已经得到改善。在 Linux 内核参数中，还有其他的一些内存调整的参数可以进行调整，但总体来说带来的收益并不大，有兴趣的读者可以再深入研究。

## 10.3 数据库调优

### 10.3.1 全局参数调整

在 postgresql.conf 配置文件中有很多参数可以灵活配置数据库的行为，本章介绍其中几个容易产生分歧或容易忽略的，对性能影响较大的参数的调整方法和原则。



在 PostgreSQL 数据库启动时，就会分配所有的共享内存，即使没有请求，共享内存也会保持固定的大小，共享内存大小由 `shared_buffers` 参数决定。当 PostgreSQL 在接收到客户端请求时，服务进程会首先在 `shared_buffers` 查找所需的数据，如果数据已经在 `shared_buffers` 中，服务进程可以在内存中进行客户端请求的处理；如果 `shared_buffers` 中没有所需数据，则会从操作系统请求数据，多数情况这些数据将从磁盘进行加载，我们知道磁盘相对内存的存取速度要慢很多，增加 `shared_buffers` 能使服务进程尽可能从 `shared_buffers` 中找到所需数据，避免去读磁盘。

在默认的 `postgresql.conf` 中，`shared_buffers` 的值都设置得很小，在 PostgreSQL 10 中，它的默认值只有 128MB，对于目前大多数的服务器硬件配置以及应对的请求量来说，这个值太过保守，建议设置大一些。由于 PostgreSQL 依赖于操作系统缓存的方式，`shared_buffers` 的值也不是越大越好，建议该参数根据不同的硬件配置，使用 `pgbench` 进行测试，得到一个最佳值。

`work_mem` 用来限制每个服务进程进行排序或 hash 时的内存分配，指定内部排序和 hash 在使用临时磁盘文件之前能使用的内存数量，它的默认值是 4MB，因为它是针对每个服务进程设置的，所以不宜设置太大。当每个进程得到的 `work_mem` 不足以排序或 hash 使用时，排序会借助磁盘临时文件，使得排序和 hash 的性能严重下降。配置该参数时，有必要了解服务器上所运行的查询的特征，如果主要运行一些小数据量排序的查询，可以不用设置过大。PostgreSQL 在排序时有 Top-N heapsort、Quick sort、External merge 这几种排序方法，如果在查询计划中发现使用了 External merge，说明需要适当增加 `work_mem` 的值。

`random_page_cost` 代表随机访问磁盘块的代价估计。参数的默认值是 4，如果使用机械磁盘，这个参数对查询计划没有影响，但现在越来越多的服务器使用固态磁盘，它也成为一个重要的参数，如果使用固态磁盘，建议将它设置为比 `seq_page_cost` 稍大即可，例如 1.5，使得查询规划器更倾向于索引扫描。

PostgreSQL 还有很多与性能相关的参数，在官方手册中对每一个参数都进行了详细的说明，读者可以进行深入研究，这里不再赘述。

### 10.3.2 统计信息和查询计划

在运行期间，PostgreSQL 会收集大量的数据库、表、索引的统计信息，查询优化器通过这些统计信息估计查询运行的时间，然后选择最快的查询路径。这些统计信息都保存在 PostgreSQL 的系统表中，这些系统表都以 `pg_stat` 或 `pg_statio` 开头。这些统计信息一类是支撑数据库系统内部方法的决策数据，例如决定何时运行 `autovacuum` 和如何解释查询计划。这些数据保存在 `pg_statistics` 中，这个表只有超级用户可读，普通用户没有权限，需要查看这些数据，可以从 `pg_stats` 视图中查询。另一类统计数据用于监测数据库级、表级、语句级的信息。本节介绍几个常用的重要系统表和系统视图。





## 1. pg\_stat\_database

数据库级的统计信息可以通过 pg\_stat\_database 这个系统视图来查看，它的定义如下：

Column	Type	Collation			Nullable	Default
datid	oid					
datname	name					
numbackends	integer					
xact_commit	bigint					
xact_rollback	bigint					
blk_s_read	bigint					
blk_s_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					
blk_read_time	double precision					
blk_write_time	double precision					
stats_reset	timestamp with time zone					

参数说明如下：

- numbackends：当前有多少个并发连接，理论上控制在 cpu 核数的 1.5 倍可以获得更好的性能；
- blk\_s\_read,blk\_s\_hit：读取磁盘块的次数与这些块的缓存命中数；
- xact\_commit, xact\_rollback：提交和回滚的事务数；
- deadlocks：从上次执行 pg\_stat\_reset 以来的死锁数量。

通过下面的查询可以计算缓存命中率：

```
SELECT blk_s_hit::float/(blk_s_read + blk_s_hit) as cache_hit_ratio FROM pg_stat_database WHERE datname=current_database();
```

缓存命中率是衡量 I/O 性能的最重要指标，它应该非常接近 1，否则应该调整 shared\_buffers 的配置，如果命中率低于 99%，可以尝试调大它的值。

通过下面的查询可以计算事务提交率：

```
SELECT xact_commit::float/(xact_commit + xact_rollback) as successful_xact_ratio FROM pg_stat_database WHERE datname=current_database();
```

事务提交率则可以知道我们应用的健康情况，它应该等于或非常接近 1，否则检查是否死锁或其他超时太多。





在 pg\_stat\_database 系统视图的字段中，除 numbackends 字段和 stats\_reset 字段外，其他字段的值是自从 stats\_reset 字段记录的时间点执行 pg\_stat\_reset() 命令以来的统计信息。建议使用者在进行优化和参数调整之后执行 pg\_stat\_reset() 命令，方便对比优化和调整前后的各项指标。有读者看到“stat”“reset”字样的命令会心存顾虑，担心执行这条命令会影响查询计划，实际上决定查询计划的是系统表 pg\_statistics，它的数据是由 ANALYZE 命令来填充，所以不必担心执行 pg\_stat\_reset() 命令会影响查询计划。

## 2. pg\_stat\_user\_tables

表级的统计信息最常用的是 pg\_stat\_user(all)\_tables 视图，它的定义如下：

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				

last\_vacuum, last\_analyze：最后一次在此表上手动执行 vacuum 和 analyze 的时间。

last\_autovacuum, last\_autoanalyze：最后一次在此表上被 autovacuum 守护程序执行 autovacuum 和 analyze 的时间。

idx\_scan, idx\_tup\_fetch：在此表上进行索引扫描的次数以及以通过索引扫描获取的行数。

seq\_scan, seq\_tup\_read：在此表上顺序扫描的次数以及通过顺序扫描读取的行数。

n\_tup\_ins, n\_tup\_upd, n\_tup\_del：插入、更新和删除的行数。

n\_live\_tup, n\_dead\_tup：live tuple 与 dead tuple 的估计数。

从性能角度来看，最有意义的数据是与索引 vs 顺序扫描有关的统计信息。当数据库可





以使用索引获取那些行时，就会发生索引扫描。另一方面，当一个表必须被线性处理以确定哪些行属于一个集合时，会发生顺序扫描。因为实际的表数据存储在无序的堆中，读取行是一项耗时的操作，顺序扫描对于大表来说是成本非常高。因此，应该调整索引定义，以便数据库尽可能少地执行顺序扫描。索引扫描与整个数据库的所有扫描的比率可以计算如下：

```
SELECT sum(idx_scan)/(sum(idx_scan) + sum(seq_scan)) as idx_scan_ratio FROM pg_stat_all_tables WHERE schemaname='your_schema';
SELECT relname, idx_scan::float/(idx_scan+seq_scan+1) as idx_scan_ratio FROM pg_stat_all_tables WHERE schemaname='your_schema' ORDER BY idx_scan_ratio ASC;
```

索引使用率应该尽可能地接近 1，如果索引使用率比较低应该调整索引。有一些很小的表可以忽略这个比例，因为顺序扫描的成本也很低。

### 3. pg\_stat\_statements

语句级的统计信息一般通过 pg\_stat\_statements、postgres 日志、auto\_explain 来获取。

开启 pg\_stat\_statements 需要在 postgresql.conf 中配置，如下所示：

```
shared_preload_libraries = 'pg_stat_statements'
pg_stat_statements.track = all
```

然后执行 CREATE EXTENSION 启用它，如下所示：

```
mydb=# CREATE EXTENSION pg_stat_statements;
CREATE EXTENSION
```

pg\_stat\_statements 视图的定义如下：

```
mydb=# \d pg_stat_statements
          View "public.pg_stat_statements"
   Column    |      Type      | Collation | Nullable | Default
-----+-----+-----+-----+-----+
userid  | oid           |           |           |
dbid    | oid           |           |           |
queryid | bigint        |           |           |
query   | text          |           |           |
calls   | bigint        |           |           |
total_time | double precision |           |           |
min_time  | double precision |           |           |
max_time  | double precision |           |           |
mean_time | double precision |           |           |
stddev_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

pg\_stat\_statements 提供了很多维度的统计信息，最常用的是统计运行的所有查询的总的调用次数和平均的 CPU 时间，对于分析慢查询非常有帮助。例如查询平均执行时间最长的 3 条查询，如下所示：

```
mydb=# SELECT calls, total_time/calls AS avg_time, left(query, 80) FROM pg_stat_statements ORDER BY 2 DESC LIMIT 3;
calls |      avg_time      |                                         left
-----+-----+-----+
 678704 | 1084.18282038266 | SELECT id, user_id, user_name, created_time, status
          FROM tbl;
 678704 | 1081.78246124378 | SELECT f.* FROM tbl_f f INNER JOIN tbl_u u ON f.id...
    126 | 365.336761904762 | SELECT tableoid, oid, proname, prolang,
          pronargs, proargtypes, prorettype, proac
(3 rows)
```

通过查询 pg\_stat\_statements 视图，可以决定先对哪些查询进行优化可获得的收益最高，对性能提升最大。执行 pg\_stat\_statements\_reset 可以重置 pg\_stat\_statements 的统计信息。

#### 4. 查看 SQL 的执行计划

执行计划，也叫作查询计划，会显示将怎样扫描语句中用到的表，例如使用顺序扫描还是索引扫描等等，以及多个表连接时使用什么连接算法来把每个输入表的行连接在一起。在 PostgreSQL 中使用 EXPLAIN 命令来查看执行计划，例如：

```
mydb=# EXPLAIN SELECT * FROM tbl;
      QUERY PLAN
-----
 Seq Scan on tbl  (cost=0.00..20.70 rows=1070 width=48)
(1 row)
```

在 EXPLAIN 命令后面还可以跟上 ANALYZE 得到真实的查询计划，例如：

```
mydb=# EXPLAIN ANALYZE SELECT * FROM tbl;
      QUERY PLAN
-----
 Seq Scan on tbl  (cost=0.00..20.70 rows=1070 width=48) (actual time=0.021..0.023
               rows=3 loops=1)
 Planning time: 0.117 ms
 Execution time: 0.058 ms
(3 rows)
```

但是需要注意的是：使用 ANALYZE 选项时语句会被执行，所以在分析 INSERT、





UPDATE、DELETE、CREATE TABLE AS 或者 EXECUTE 命令的查询计划时，应该使用一个事务来执行，得到真正的查询计划后对该事务进行回滚，就会避免因为使用 ANALYZE 选项而修改了数据，例如：

```
mydb=# BEGIN;
BEGIN
mydb=# EXPLAIN ANALYZE UPDATE tbl SET ival = ival * 10 WHERE id = 1;
        QUERY PLAN
-----
Update on tbl  (cost=0.15..8.17 rows=1 width=54) (actual time=0.159..0.159 rows=0
    loops=1)
    -> Index Scan using tbl_pkey on tbl  (cost=0.15..8.17 rows=1 width=54) (actual
        time=0.046..0.047 rows=1 loops=1)
            Index Cond: (id = 1)
Planning time: 4.237 ms
Execution time: 0.315 ms
(5 rows)
mydb=# ROLLBACK;
ROLLBACK
```

在阅读查询计划时，有一个简单的原则：从下往上看，从右往左看。例如上面的例子，从下往上看最后一行的内容是 Execution time: 0.315 ms，是这条语句的实际执行时间是 0.315 ms；往上一行的内容是 Planning time: 4.237 ms，是这条语句的查询计划时间 4.237 ms；往上一行，一个箭头在上一行的右侧缩进处，表示先使用 tbl 表上的 tbl\_pkey 进行了 Index Scan，然后到最上面一行，在 tbl 表上执行了 Update。在每行计划中，都有几项值，(cost=0.00..xxx) 预估该算子开销有多么“昂贵”。“昂贵”按照磁盘读计算。这里有两个数值：第一个表示算子返回第一条结果集最快需要多少时间；第二个数值（通常更重要）表示整个算子需要多长时间。开销预估中的第二项 (rows=xxx) 表示 PostgreSQL 预计该算子会返回多少条记录。最后一项 (width=1917) 表示结果集中一条记录的平均长度（字节数），由于使用了 ANALYZE 选项，后面还会有实际执行的时间统计。

除了 ANALYZE 选项，还可以使用 COSTS、BUFFERS、TIMING、FORMAT 这些选项输出比较详细的查询计划，例如：

```
mydb=# EXPLAIN (ANALYZE on, TIMING on , VERBOSE on, BUFFERS on) SELECT * FROM tbl
      WHERE id = 10;
        QUERY PLAN
-----
Index Scan using tbl_pkey on public.tbl  (cost=0.15..8.17 rows=1 width=48) (actual
    time=0.015..0.015 rows=0 loops=1)
    Output: id, ival, description, created_time
    Index Cond: (tbl.id = 10)
    Buffers: shared hit=1
Planning time: 0.177 ms
Execution time: 0.065 ms
(6 rows)
```





使用 EXPLAIN 的选项（ANALYZE、COSTS、BUFFERS、TIMING、VERBOSE）可以帮助开发人员获取非常详细的查询计划，但是有时候我们会有一些需要高度优化的需求，或者是一些语句的查询计划提供的信息不能完全判断语句的优劣，这时候还可以使用 session 级的 log\_xxx\_stats 来判断问题。PostgreSQL 在 initdb 后，log\_statement\_stats 参数默认是关闭的，因为打开它会在执行每条命令的时候，执行大量的系统调用来收集资源消耗信息，所以在生产环境中也应该关闭它，一般都在 session 级别使用它。

在 postgresql.conf 中 有 log\_parser\_stats、log\_planner\_stats 和 log\_statement\_stats 这几个选项，默认值都是 off，其中 log\_parser\_stats 和 log\_planner\_stats 这两个参数为一组，log\_statement\_stats 为一组，这两组参数不能全部同时设置为 on。

查看 parser 和 planner 的系统资源使用的查询计划，如下所示：

```
mydb=# set client_min_messages = log;
mydb=# set log_parser_stats = on;
mydb=# set log_planner_stats = on;
```

运行 EXPLAIN ANALYZE 查看查询计划，如下所示：

```
mydb=# EXPLAIN ANALYZE select * from tbl limit 10;
LOG:  PARSE STATISTICS
DETAIL: ! system usage stats:
!    0.000060 elapsed 0.000000 user 0.000000 system sec
!    [0.061990 user 0.014997 sys total]
!    0/0 [600/0] filesystem blocks in/out
!    0/0 [0/2640] page faults/reclaims, 0 [0] swaps
!    0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!    0/0 [55/1] voluntary/involuntary context switches
LOG:  PARSE ANALYSIS STATISTICS
DETAIL: ! system usage stats:
!    0.000086 elapsed 0.000000 user 0.000000 system sec
!    [0.061990 user 0.014997 sys total]
!    0/0 [600/0] filesystem blocks in/out
!    0/0 [0/2640] page faults/reclaims, 0 [0] swaps
!    0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!    0/0 [55/1] voluntary/involuntary context switches
LOG:  REWRITER STATISTICS
DETAIL: ! system usage stats:
!    0.000001 elapsed 0.000000 user 0.000000 system sec
!    [0.061990 user 0.014997 sys total]
!    0/0 [600/0] filesystem blocks in/out
!    0/0 [0/2640] page faults/reclaims, 0 [0] swaps
!    0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
!    0/0 [55/1] voluntary/involuntary context switches
LOG:  PLANNER STATISTICS
DETAIL: ! system usage stats:
!    0.000165 elapsed 0.001000 user 0.000000 system sec
!    [0.063990 user 0.014997 sys total]
!    0/0 [600/0] filesystem blocks in/out
```





```
! 0/0 [0/2640] page faults/reclaims, 0 [0] swaps
! 0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
! 0/0 [58/1] voluntary/involuntary context switches
QUERY PLAN
-----
Limit  (cost=0.00..0.43 rows=10 width=224) (actual time=0.028..0.034 rows=10 loops=1)
-> Seq Scan on tbl  (cost=0.00..64771378.60 rows=1496764160 width=224)
   (actual time=0.025..0.027 rows=10 loops=1)
Planning time: 0.198 ms
Execution time: 0.083 ms
(4 rows)
```

查看查询的系统资源使用，如下所示：

```
mydb=# set client_min_messages = log;
mydb=# set log_parser_stats = off;
mydb=# set log_planner_stats = off;
mydb=# set log_statement_stats = on;
mydb=# EXPLAIN ANALYZE select * from tbl limit 10;
LOG:  QUERY STATISTICS
DETAIL: ! system usage stats:
        0.000603 elapsed 0.000000 user 0.000000 system sec
        [0.065989 user 0.014997 sys total]
        0/0 [160/0] filesystem blocks in/out
        0/0 [0/2640] page faults/reclaims, 0 [0] swaps
        0 [0] signals rcvd, 0/0 [0/0] messages rcvd/sent
        0/0 [62/1] voluntary/involuntary context switches
QUERY PLAN
-----
Limit  (cost=0.00..0.43 rows=10 width=224) (actual time=0.027..0.033 rows=10 loops=1)
-> Seq Scan on tbl  (cost=0.00..64771378.60 rows=1496764160 width=224) (actual
   time=0.026..0.026 rows=10 loops=1)
Planning time: 0.154 ms
Execution time: 0.082 ms
(4 rows)
```

查看 parser 和 planner 的系统资源使用情况的查询计划输出内容很多，在 DETAIL: ! system usage stats：之后的前两行显示查询使用的用户和系统 CPU 时间和已经消耗的时间。第 3 行显示存储设备（而不是内核缓存）中的 I/O。第 4 行涵盖内存页面错误和回收的进程地址空间。第 5 行显示信号和 IPC 消息活动。第 6 行显示进程上下文切换。

### 10.3.3 索引管理与维护

索引在所有关系型数据库的性能方面都扮演着极其重要的角色。针对不同的使用场景，PostgreSQL 有许多种索引类型来应对，例如 B-tree、Hash、GiST、SP-GiST、GIN、BRIN 和 Bloom 等等。最常用最常见的索引类型是 B-tree。B-tree 索引在执行 CREATE INDEX 命令时是默认的索引类型。B-tree 索引通常用于等值和范围查询；Hash 索引只能处理简单等值查询；GIN 索引是适合于包含多个组成值的数据值，例如数组；GiST 索引并不是一种单



独的索引，而是一个通用的索引接口，可以使用 GiST 实现 B-tree、R-tree 等索引结构，在 PostGIS 中使用最广泛的就是 GiST 索引。除了支持众多类型的索引，PostgreSQL 也支持唯一索引、表达式索引、部分索引和多列索引，B-tree、GiST、GIN 和 BRIN 索引都支持多列索引，最多可以支持 32 个列的索引。

相同的查询，有时候会使用索引，但有时候会使用顺序扫描，这种情况也是存在的。大多数时候是因为顺序扫描有可能比索引扫描所扫描的块更多，遇到这种情况还需要仔细的分析。

在 PostgreSQL 中执行 CREATE INDEX 命令时，可以使用 CONCURRENTLY 参数并行创建索引，使用 CONCURRENTLY 参数不会锁表，创建索引过程中不会阻塞表的更新、插入、删除操作。由于 PostgreSQL 的 MVCC 内部机制，当运行大量的更新操作后，会有“索引膨胀”的现象，这时候可以通过 CREATE INDEX CONCURRENTLY 在不阻塞查询和更新的情况下，在线重新创建索引，创建好新的索引之后，再删除原先有膨胀的索引，减小索引尺寸，提高查询速度。对于主键也可以使用这种方式进行重建，重建方法如下：

```
mydb=# CREATE UNIQUE INDEX CONCURRENTLY ON mytbl USING btree(id);
CREATE INDEX
```

这时可以看到 id 字段上同时有两个索引 mytbl\_pkey 和 mytbl\_id\_idx，如下所示：

```
mydb=# SELECT schemaname,relname,indexrelname,pg_relation_size(indexreloid) AS
       index_size,idx_scan,idx_tup_read,idx_tup_fetch FROM pg_stat_user_indexes
      WHERE indexrelname IN (SELECT indexname FROM pg_indexes WHERE schemaname =
        'public' AND tablename = 'mytbl');
schemaname | relname | indexrelname | index_size | idx_scan | idx_tup_read | idx_
           tup_fetch
-----
public   | mytbl  | mytbl_pkey  | 223051776 | 1403532  |    1413850  |     1403532
public   | mytbl  | mytbl_id_idx | 222887936 |         0  |          0  |         0
(2 rows)
```

开启事务删除主键索引，同时将第二索引更新为主键的约束，如下所示：

```
mydb=# BEGIN;
BEGIN
mydb=# ALTER TABLE mytbl DROP CONSTRAINT mytbl_pkey;
ALTER TABLE
mydb=# ALTER TABLE mytbl ADD CONSTRAINT mytbl_id_idx PRIMARY KEY USING INDEX
       mytbl_id_idx;
ALTER TABLE
mydb=# END;
COMMIT
```

检查表索引，现在只有第二索引了，如下所示：

```
mydb=# SELECT schemaname,relname,indexrelname,pg_relation_size(indexreloid) AS
       index_size,idx_scan,idx_tup_read,idx_tup_fetch FROM pg_stat_user_indexes WHERE
      indexrelname IN (SELECT indexname FROM pg_indexes WHERE#
```

```

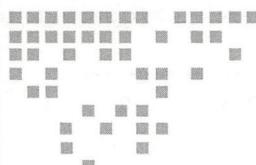
schema_name | relname | indexrelname | index_size | idx_scan | idx_tup_read |
idx_tup_fetch.
-----
public      | mytbl   | mytbl_id_idx | 222887936 | 0       | 0       | 0
(1 row)

```

这样就完成了主键索引的重建，对于大规模的数据库集群，可以通过 pg\_repack 工具进行定时的索引重建。

## 10.4 本章小结

本章简单介绍了服务器硬件、操作系统配置对性能的影响，介绍了一些常用的 Linux 监控性能工具，并着重介绍了对性能影响较大的几个方面：I/O 调度算法、预读参数、Swap、透明大页、NUMA 等，以及它们的调整原则和方法。在数据库层面介绍了几个常用和容易被忽略的参数，并简单介绍了数据库级别、表级别、查询级别统计信息的系统视图，以及如何得到最详细的查询计划的方法。最后分享了一些索引方面的管理和维护建议。除了文中介绍的内容，定时进行 VACUUM 操作和 ANALYZE 操作，在业务低谷时段定时进行 VACUUM FREEZE 操作，及时处理慢查询，硬件的监控维护也很重要，性能调优不是一次性任务，也不能在性能表现已经很差的时候才去做，好的监控系统，历史数据的分析，慢查询的优化都需要不断完善，才能保障业务系统稳定高效运转。



## 第 11 章

# 基准测试与 pgbench

在数据库服务器的硬件、软件环境中建立已知的性能基准线称为基准测试，是数据库管理员和运维人员需要掌握的一项基本技能。设计优良的基准测试有助于数据库选型，了解数据库产品的性能特点，提升产品质量。根据测试目的的不同，可针对压力、性能、最大负载进行专门测试，或综合测试。通过定量的、可复现的、能对比的方法衡量系统的吞吐量，也可以对新硬件的实际性能和可靠性进行测试，或在生产环境遇到问题时，在测试环境复现问题。本章将讨论 PostgreSQL 数据库和基于 PostgreSQL 数据库开发的应用系统的测试方法和常见的测试工具，并会着重讲解 PostgreSQL 内置的 pgbench 测试工具。

## 11.1 关于基准测试

我们的软件系统都是运行在一定环境中的，例如软件运行的操作系统、文件系统和硬件。这些都受到一些关键因素影响：

- 硬件，如服务器配置、CPU、内存、存储，通常硬件越高级，系统的性能越好；
- 网络，带宽不足也会严重限制系统整体性能表现；
- 负载，不同的用户数，不同的数据量对系统的性能影响也非常大；
- 软件，不同的数据库在不同的操作系统、不同的应用场景下性能表现有很大的不同。

在现在流行的应用中，应用服务器、网络、缓存都比较容易进行水平扩展，达到提高性能和吞吐量的目的，但关系型数据库管理系统的水平扩展能力受到很多因素限制，一般只能通过增加缓存层、分库分表及读写分离来减轻面临的压力，对多数使用关系型数据库的应用系统，瓶颈主要在数据库，因此数据库层的基准测试尤为重要。整个应用系统的测

试和评估是一项非常复杂的工作，这里我们只讨论 PostgreSQL 数据库的基准测试。

### 11.1.1 基准测试的常见使用场景

基准测试可用于测试不同的硬件和应用场景下的数据库系统配置是否合理。例如更换新型的磁盘对当前系统磁盘 I/O 能力不足的问题是否有所帮助？升级了操作系统内核版本是否有性能提升？不同的数据库版本和不同的数据库参数配置的表现是怎样的？

通过模拟更高的负载，可以预估压力增加可能带来的瓶颈，也可以对未来的业务增长规划有所帮助。

重现系统中高负载时出现的错误或异常。当生产环境在高负载情况下出现异常行为时，很多时候并不能即时捕捉到异常的原因，在测试环境中去模拟出现异常时的高负载场景，进行分析并解决问题，是很好的办法。

新系统上线前，大致模拟上线之后的用户行为以进行测试，根据测试的结果对系统设计、代码和数据库配置进行调整，使新系统上线即可达到较好的性能状态。

模拟较高的负载，知道系统在什么负载情况下将无法正常工作，也就是通常说的“容量规划”。作为容量规划的参考，需要注意的是不能以基准测试的结果简单地进行假设。因为数据库的状态一直在改变，随着时间的推移或业务的增长，数据量、请求量、并发量以及数据之间的关系都在发生着变化，还可能有很多功能特性的变化，有一些新功能的影响可能远远大于目前功能的压力总量，对容量规划只能做大概的评估，这是数据库系统的基准相比于其他无状态系统测试的不同点。

### 11.1.2 基准测试衡量指标

通常数据库的基准测试最关键的衡量指标有：吞吐量（Throughput）、响应时间（RT）或延迟（Latency）和并发量。

吞吐量衡量数据库的单位时间内的事务处理能力，常用的单位是 TPS（每秒事务数）。响应时间或延迟，描述操作过程里用来响应服务的时间，根据不同的应用可以使用分钟、秒、毫秒和微秒作为单位。通常还会根据响应时间的最大值、最小值以及平均值做分组统计，例如 90% 的运行周期内的响应时间是 1 毫秒，10% 的运行周期内响应时间是 5 毫秒，这样可以得出相对客观的测试结果。并发量是指同时工作的连接数。在不同的测试场景，需要关注的指标也会不同，分析测试结果时，吞吐量、响应时间、并发量是必须关注的三个基本要素。

### 11.1.3 基准测试的原则

面对一个复杂的系统，在测试之前应该先明确测试的目标。在一次测试中不可能将系统的各方面都测试得很清楚，每个测试尽量目标单一，测试方法简单。例如新增加了一个索引，需要测试这个索引对性能的影响，那么我们的测试只针对这一个查询，关注索引调

整前后这个查询的响应时间和吞吐量的变化，但如果同时还做了很多其他可能影响到测试结果的变更，就可能无法得出明确的测试结果。测试过程应该尽量持续一定时间，如果测试时间太短，则可能因为没有缓存而得到不准确的测试结果；在测试过程中应该尽量接近真实的应用场景，并且应该尽可能地多收集系统状态，例如参数配置、CPU 使用率、I/O、网络流量统计等，即使这些数据目前可能不需要，但也应该先保留下来，避免测试结果缺乏依据。

每轮测试结束后，都应该详细记录当时的配置和结果，并尽量将这些信息保存为容易使用脚本或工具分析的格式。

实际测试的时候，并不会很顺利，有时候得到的测试结果可能与其他几次的测试结果出入很大，这时也应该仔细分析原因，例如查看错误日志等。

## 11.2 使用 pgbench 进行测试

TPC（事务处理性能委员会：Transaction Processing Performance Council，<http://www.tpc.org>）已经推出了 TPC-A、TPC-B、TPC-C、TPC-D、TPC-E、TPC-W 等基准程序的标准规范，其中 TPC-C 是经典的衡量在线事务处理（OLTP）系统性能和可伸缩性的基准测试规范，还有比较新的 OLTP 测试规范 TPC-E。常见的开源数据库的基准测试工具有 benchmarksql、sysbench 等，PostgreSQL 自带运行基准测试的简单程序 pgbench。pgbench 是一个类 TPC-B 的基准测试工具，可以执行内置的测试脚本，也可以自定义脚本文件。

### 11.2.1 pgbench 的测试结果报告

在 pgbench 运行结束后，会输出一份测试结果的报告，典型的输出如下所示：

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 100
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 2.557 ms
tps = 391.152261 (including connections establishing)
tps = 399.368200 (excluding connections establishing)
```

transaction type 行记录本次测试所使用的测试类型；

scaling factor 记录 pgbench 在初始化时设置的数据量的比例因子；

query mode 是测试时指定的查询类型，包括 simple 查询协议、extended 查询协议或 prepared 查询协议；

number of clients 是测试时指定的客户端数量；

number of threads 是测试时指定的每个客户端的线程数；

number of transactions per client 是测试时指定的每个客户端运行的事务数；

number of transactions actually processed 是测试结束时实际完成的事务数和计划完成的事务数，计划完成的事务数只是客户端数量乘以每个客户端的事务数的值。如果测试成功结束，实际完成的事务数应该和计划完成的事务数相等，如果有事务执行失败，则只会显示实际完成的事务数。

latency average 是测试过程中的平均响应时间；

最后两行 TPS 的值分别是包含和不包含建立连接开销的 TPS 值。

## 11.2.2 通过内置脚本进行测试

### 1. 初始化测试数据

pgbench 的内嵌脚本需要 4 张表：pgbench\_branches、pgbench\_tellers、pgbench\_accounts 和 pgbench\_history。使用 pgbench 初始化测试数据，pgbench 会自动去创建这些表并生成测试数据。在初始化过程中，如果数据库中存在和这些表同名的数据，pgbench 会删除这些表重新进行初始化。

pgbench 初始化语法如下所示：

```
pgbench -i [OPTION]... [DBNAME]
```

pgbench 初始化选项如下所示：

-i, --initialize	进入初始化模式；
-F, --fillfactor=NUM	设置创建表时，数据块的填充因子，这个值的取值范围为10到100之间，可以为小数，默认值是100，设置小于100的值对于UPDATE的性能会有一定程度的提升；
-n, --no-vacuum	初始化结束后不执行VACUUM操作；
-q, --quiet	初始化生成测试数据的过程中，默认会在每100000行时打印一条消息，切换到静默模式后，则只在每5秒打印一条消息，开启该选项可以避免在生成大量数据时打印过多的消息；
-s, --scale=NUM	可以将这些表理解为公司的账户数据，出纳会在账户中记录资金数据的流入流出，同时系统会记录到操作历史表中。该参数是生成数据的比例因子，默认值为1。当它的值是1时，只创建一家公司的账户，当它的值为k时，生成k个公司的数据，每个公司默认生成的测试数据量如下：
pgbench_branches	1k
pgbench_tellers	10k
pgbench_accounts	100000k
pgbench_history	0
--foreign-keys	在上述的4张测试表之间创建外键约束。
--index-tablespace=TABLESPACE	在指定的表空间创建索引
--tablespace=TABLESPACE	在指定的表空间创建表
--unlogged-tables	把上述4张测试表创建为UNLOGGED表

初始化测试数据的命令如下所示：

```
[postgres@ghost2 ~]$ /usr/pgsql-10/bin/pgbench -i -s 2 -F 80 -h pghost1 -p 1921
-U pguser -d mydb
creating tables...
100000 of 200000 tuples (50%) done (elapsed 0.02 s, remaining 0.02 s)
200000 of 200000 tuples (100%) done (elapsed 0.16 s, remaining 0.00 s)
vacuum...
set primary keys...
done.
```

## 2. 使用 pgbench 内置脚本进行测试

pgbench 的内置测试脚本有 tpcb-like、simple-update 和 select-only 三种。可以通过以下命令查看当前版本的 pgbench 包含哪些集成的测试脚本：

```
[postgres@ghost2 ~]$ /usr/pgsql-10/bin/pgbench -b list
Available builtin scripts:
  tpcb-like
  simple-update
  select-only
```

tpcb-like 执行的脚本内容是一个包含 SELECT、UPDATE、INSERT 的事务：

```
BEGIN;
  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid,
    :aid, :delta, CURRENT_TIMESTAMP);
END;
```

simple-update 执行的脚本如下所示：

```
BEGIN;
  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid,
    :aid, :delta, CURRENT_TIMESTAMP);
END;
```

select-only 执行的脚本如下所示：

```
BEGIN;
  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
END;
```

可以使用下列参数设置内置脚本测试的方式：

```
-b scriptname[@weight]
--builtin = scriptname[@weight]
```

scriptname 参数指定使用哪一种脚本进行测试，默认使用 tpcb-like 这一种测试脚本。

例如使用 simple-update 进行测试，代码如下所示：

```
[postgres@pghost2 ~]$ /usr/pgsql-10/bin/pgbench -b simple-update -h pgghost1 -p 1921 -U pguser mydb
starting vacuum...end.
transaction type: <builtin: simple update>
scaling factor: 100
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 1.631 ms
tps = 613.238093 (including connections establishing)
tps = 631.101768 (excluding connections establishing)
```

还可以选择 3 种内置脚本混合进行测试，并在脚本名称后面加上 @ 符号，@ 符号后面加一个脚本运行比例的权重的整数值，例如使用 simple-update 和 select-only 两种内置脚本，并且以 2:8 的比例混合进行测试，代码如下所示：

```
[postgres@pghost2 ~]$ /usr/pgsql-10/bin/pgbench -b simple-update@2 -b select-only@8 -b tpcb@0 -h pgghost1 -p 1921 -U pguser mydb
starting vacuum...end.
transaction type: multiple scripts
scaling factor: 100
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 0.617 ms
tps = 1621.779592 (including connections establishing)
tps = 1753.156910 (excluding connections establishing)
SQL script 1: <builtin: TPC-B (sort of)>
- weight: 0 (targets 0.0% of total)
- 0 transactions (0.0% of total, tps = 0.000000)
- latency average = -nan ms
- latency stddev = -nan ms
SQL script 2: <builtin: select only>
- weight: 8 (targets 80.0% of total)
- 9 transactions (90.0% of total, tps = 1459.601633)
- latency average = 0.439 ms
- latency stddev = 0.137 ms
SQL script 3: <builtin: simple update>
- weight: 2 (targets 20.0% of total)
- 1 transactions (10.0% of total, tps = 162.177959)
- latency average = 1.738 ms
- latency stddev = 0.000 ms
```

在 @ 符后面的测试脚本名称还可以在名称不冲突的情况下，使用名称的前几个字母进行简写，例如上面的例子就可以将 tpcb-like 简写为 tpcb 甚至是 t，simple 和 select 都是 s 开

头，就不能简写为 s 了，但可以简写为 si 和 se，否则会有如下错误：

```
ambiguous builtin name: 2 builtin scripts found for prefix "s"
```

-b simple-update 可以使用 -N 或 --skip-some-updates 参数简写，-b select-only 可以用 -S 或 --select-only 简写，但是如果在混合测试时，使用这种简写方式，则不能指定它们各自的占比。

使用内置脚本混合测试，最终输出的结果除了常规的报告项之外，还会输出每个测试脚本的实际占比，以及每个类型的测试的平均延时、TPS 等等更加详细的值。使用混合方式的测试，可以方便模拟不同的读写比。

### 11.2.3 使用自定义脚本进行测试

仅使用 pgbench 内置的测试脚本，会有很多限制，例如不能更真实地模拟真实世界的数据量，数据结构和运行的查询，还有其他的局限性。pgbench 支持从一个文件中读取事务脚本来替换默认的事务脚本，达到运行自定义测试场景的目的。

#### 1. 创建测试表

代码如下所示：

```
CREATE TABLE tbl
(
    id SERIAL PRIMARY KEY,
    ival INT
);
```

#### 2. 运行自定义脚本

在 pgbench 命令中使用 -f 参数运行自定义的脚本，举例如下：

```
[postgres@pghost2 ~]$ echo "SELECT id,ival FROM tbl ORDER BY id DESC LIMIT 10;" > bench_script_for_select.sql
[postgres@pghost2 ~]$ /usr/pgsql-10/bin/pgbench -f bench_script_for_select.sql -h pgghost1 -p 1921 -U pguser mydb
transaction type: bench_script_for_select.sql
...
tps = 3448.671865 (including connections establishing)
tps = 4097.559616 (excluding connections establishing)
```

在测试过程中，通常会使用变量作为 SQL 语句的参数传入。在脚本中可以使用类似于 psql 的以反斜杠开头的元命令和内建函数定义变量。pgbench 自定义脚本支持的元命令有：

\sleep number [ us|ms|s]：每执行一个事务暂停一定时间，单位可以是微秒、毫秒和秒，如果不写单位，默认使用秒。

\set varname expression：用于设置一个变量，元命令、参数和参数的值之间用空格分开。在 pgbench 中定义了一些可以在元命令中使用的函数，例如刚才我们使用到的 random(int,int) 函数。

例如自定义一个脚本，在tbl表的ival字段中插入一个随机数，代码如下所示：

```
[postgres@pghost2 ~]$ cat bench_script_for_insert.sql
\sleep 500 ms
\set ival random(1, 100000)
INSERT INTO tbl(ival) VALUES (:ival);
```

运行pgbench使用该脚本进行测试，如下所示：

```
[postgres@pghost2 ~]$ /usr/pgsql-10/bin/pgbench -f bench_script_for_insert.sql -h
pghost1 -p 1921 -U pguser mydb
starting vacuum...end.
transaction type: bench_script_for_insert.sql
...
latency average = 501.291 ms
number of transactions actually processed: 14846
...
```

检查测试表，如下所示：

```
mydb=# SELECT COUNT(*) FROM tbl;
 count
-----
 14846
(1 row)

mydb=# SELECT id,ival FROM tbl ORDER BY id DESC LIMIT 3;
 id | ival
-----+-----
 14846 | 95018
 14845 | 88153
 14844 | 21896
(3 rows)
```

可以看到INSERT成功地插入了预期的值，共14846条记录。从pgbench输出的测试报告看，平均延迟(latency average)等于500多毫秒，成功执行了14846个事务。

和使用内置脚本一样，可以在-f的参数值后加上@符号再加上权重，以不同比例运行多个自定义的脚本。权重值并不是一个百分比的值，而是一个相对固定的数值，例如第一个脚本运行3次，第二个脚本运行10次，下面是一个在多个测试脚本中加入权重值选项进行测试的例子：

```
[postgres@pghost2 ~]$ /usr/pgsql-10/bin/pgbench -T 60 -f bench_script_for_insert.
sql@3 -f bench_script_for_insert.sql@10 -h pghost1 -p 1921 -U pguser mydb
starting vacuum...end.
transaction type: multiple scripts
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
duration: 60 s
number of transactions actually processed: 176523
```

```

latency average = 0.340 ms
tps = 2942.048332 (including connections establishing)
tps = 2942.071422 (excluding connections establishing)
SQL script 1: bench_script_for_insert.sql
  - weight: 3 (targets 23.1% of total)
  - 40921 transactions (23.2% of total, tps = 682.016280)
  - latency average = 0.340 ms
  - latency stddev = 0.035 ms
SQL script 2: bench_script_for_insert.sql
  - weight: 10 (targets 76.9% of total)
  - 135602 transactions (76.8% of total, tps = 2260.032052)
  - latency average = 0.340 ms
  - latency stddev = 0.038 ms

```

## 11.2.4 其他选项

### 1. 模拟的客户端数量和连接方式

-c 参数指定模拟的客户端的数量，也就是并发数据库的连接数量，默认值为 1。-c 参数指定是否为每个事务创建一个新的连接，如果每次都创建新的连接则性能会明显下降很多，测试连接池性能时这个参数比较有用。

例如模拟 4 个客户端连接如下所示：

```
[postgres@pghost2 ~]$ /usr/pgsql-10/bin/pgbench -c 4 -h pgghost1 -p 1921 -U pguser mydb
...
number of clients: 4
...
latency average = 4.826 ms
tps = 828.781956 (including connections establishing)
tps = 894.930906 (excluding connections establishing)
```

每个事务创建新的连接，如下所示：

```
[postgres@pghost2 ~]$ /usr/pgsql-10/bin/pgbench -c 4 -C -h pgghost1 -p 1921 -U
pguser mydb
...
number of clients: 4
...
latency average = 19.627 ms
tps = 203.797152 (including connections establishing)
tps = 256.799857 (excluding connections establishing)
```

如果在多线程 CPU 上进行测试，还可以指定 -j 参数，将模拟的客户端平均分配在各线上。

### 2. 单次测试的运行时间

单次测试运行的时间由两种测试方式决定：

- -T seconds 或 --time=seconds 参数用来设置测试的秒数；例如需要测试 1 小时，在测试命令行中增加参数 -T 3600 即可。

- ❑ -t transactions 或 --transactions=transactions 参数指定每个客户端运行多少个固定数量的事务就结束本次测试，默认为 10 个。

这两种方式每次只能使用一种，要么指定准确的测试时间，要么指定一共执行多少个事务后结束。

如果这两个参数都没有指定，那么 pgbench 默认使用第二种方式。例如：

```
[postgres@pghost2 ~]$ /usr/pgsql-10/bin/pgbench -c 4 -h pgghost1 -p 1921 -U pguser mydb
...
number of clients: 4
...
number of transactions actually processed: 40/40
...
```

由于只指定了 4 个客户端，没有指定运行时间或总的执行事务个数，所以测试报告中“实际执行成功的事务数”和“期望执行的事务数”是 40。

### 3. 用固定速率运行测试脚本

使用 -R 可以用固定速率执行事务，单位是 TPS。如果给定的既定速率的值大于最大可能的速率，则该速率限制不会影响结果。

### 4. 超出阈值的事务的报告

使用 -L 参数设置一个阈值，对超过这个阈值的事务进行独立报告和计数，单位是毫秒。例如：

```
[postgres@pghost2 ~]$ /usr/pgsql-10/bin/pgbench -T 10 -L 1 -c 8 -j 8 -f bench_
script_for_select.sql@3 -f bench_script_for_update.sql@10 -h pgghost1 -p 1921
-U pguser mydb
...
number of transactions actually processed: 129234
number of transactions above the 1.0 ms latency limit: 2226 (1.722 %)
...
```

从上面的报告可以看出，超过 1 毫秒阈值的事务有 2226 个，占到实际执行事务总数的 1.722%。

### 5. 输出选项

pgbench 有丰富的测试结果报告的输出格式。

使用 -d 参数可以输出 debug 信息，通常不使用它。

使用 -P 参数，可以每隔一段时间输出一次测试结果，例如每隔 2 秒输出一次测试结果，如下所示：

```
[postgres@pghost2 ~]$ /usr/pgsql-10/bin/pgbench -P 2 -T 7200 -c 8 -j 8 -f bench_
script_for_select.sql@10 -f bench_script_for_update.sql@3 -h pgghost1 -p 1921
-U pguser mydb
...
progress: 2.0 s, 20448.0 tps, lat 0.390 ms stddev 0.160
```



```
...
progress: 8.0 s, 25129.4 tps, lat 0.318 ms stddev 0.108
progress: 10.0 s, 25064.1 tps, lat 0.319 ms stddev 0.116
...
```

-l 或 --log 将每一个事务执行的时间记入一个名称为 “pgbench\_log.n” 的日志文件中，如果使用 -j 参数指定使用多个线程，则会生成名称为 “pgbench\_log.n.m” 的日志文件。其中 n 是测试时 pgbench 的 PID，m 是从 1 开始的线程的序号，pgbench\_log 是默认的日志文件的前缀，如果希望自定义前缀，使用 --log-prefix prefix\_name 选项。例如：

```
[postgres@pghost2 ~]$ /usr/pgsql-10/bin/pgbench -T 10 -l --log-prefix=custom -c
6 -j 2 -f bench_script_for_insert.sql@3 -f bench_script_for_insert.sql@10 -h
pghost1 -p 1921 -U pguser mydb
```

测试结束之后，在运行 pgbench 的目录会生成 custom.151940 和 custom2.151940.2 的两个日志文件。它们的内容格式如下所示：

```
client_id transaction_no time script_no time_epoch time_us [ schedule_lag ]
5 17 1294 1 1515153407 106478
4 15 322 0 1515152900 291756
3 7 357 0 1515152900 291768
...
4 16 334 0 1515152900 292090
...
```

其中 client\_id 为客户端的序号 id；transaction\_no 为事务的序号；time 是该事务所花费的时间，单位为微秒；在使用了多个脚本时，script\_no 标识该事务使用的是哪个脚本；time\_epoch/time\_us 是一个 Unix 纪元格式的时间戳以及一个显示事务完成时间的以微秒计的偏移量（适合于创建一个带有分数秒的 ISO 8601 时间戳）。分析测试结果，最好能够将测试结果生成为直观的图表，通常可以借助电子表格。

## 11.3 本章小结

本章我们讨论了什么是基准测试，以及基准测试的常见使用场景，确立了测试的原则，明确了测试结果的衡量方法，着重介绍了 PostgreSQL 内置的测试工具 pgbench，详细介绍了如何使用 pgbench 的内置脚本和自定义脚本进行测试，并以一系列简单的例子演示了 pgbench 的用法。pgbench 有丰富的测试组合方式，熟练掌握它对日常的性能测试、压力测试等工作会有很大帮助。