

32 | 当Postgres磁盘读引起I/O高的时候，应该怎么办？

2020-03-04 高楼

性能测试实战30讲

[进入课程 >](#)



讲述：高楼

时长 25:55 大小 23.74M



在性能分析的人眼里，性能瓶颈就是性能瓶颈。无论这个性能瓶颈出现在代码层、操作系统层、数据库层还是其他层，最终的目的只有一个结果：解决掉！

有人可能会觉得这种说法过于霸道。

事实上，我要强调的性能分析能力，是一套分析逻辑。在这一套分析逻辑中，不管是操作系统、代码还是数据库等，所涉及到的都只是基础知识。如果一个人掌握这些内容，那确实不现实，但如果是对一个性能团队的要求，我觉得一点也不高。



在性能测试和性能分析的项目中，没有压力发起，就不会有性能瓶颈，也就谈不上性能分析了。所以每个问题的前提，都是要有压力。

但不是所有的压力场景都合理，再加上即使压力场景不合理，也能压出性能瓶颈，这就会产生一种错觉：似乎一个错误的压力场景也是有效的。

我是在介入一个项目时，首先会看场景是否有效。如果无效，我就不会下手去调了，因为即使优化好了，可能也给不出生产环境应该如何配置的结论，那工作就白做了。

所以要先调场景。

我经常会把一个性能测试项目里的工作分成两大阶段：

整理阶段

在这个阶段中，要把之前项目中做错的内容纠正过来。不止有技术里的纠正，还有从上到下沟通上的纠正。

调优阶段

这才真是干活的阶段。

在这个案例中，同样，我还是要表达一个分析的思路。

案例问题描述

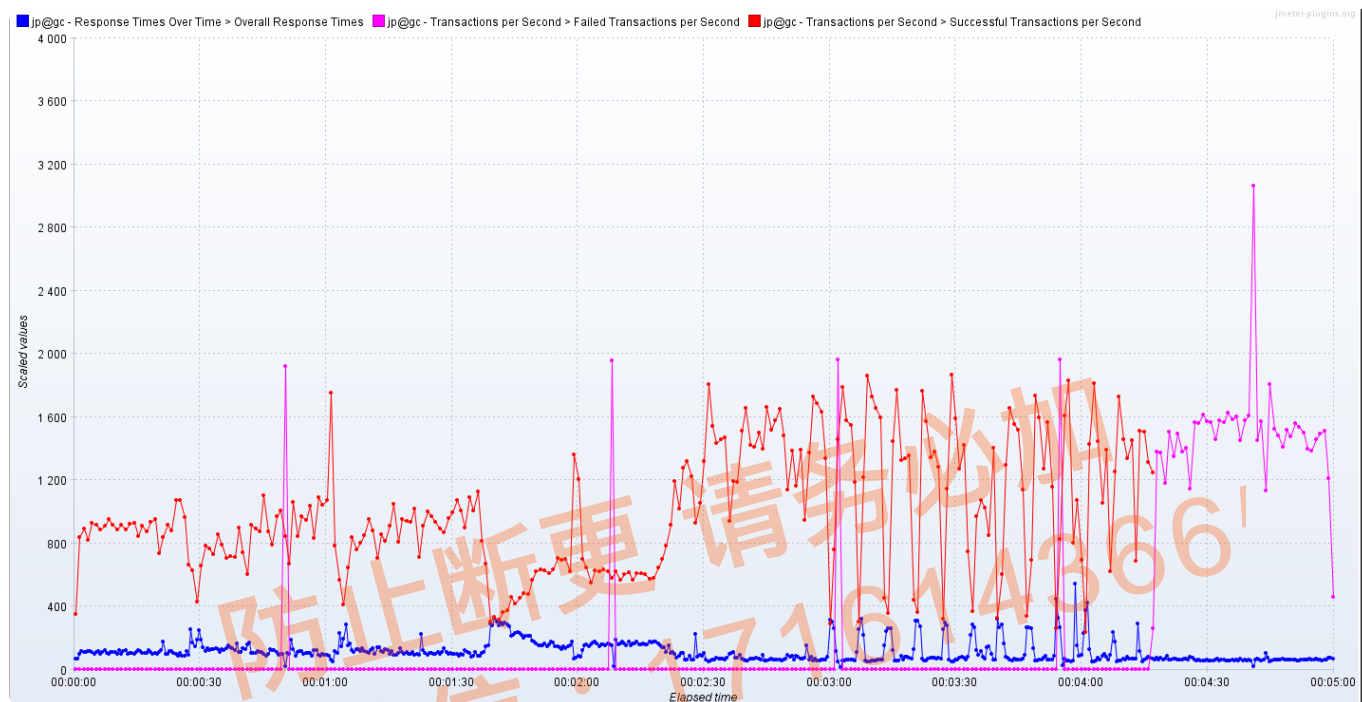
这是一个性能从业人员问的问题：为什么这个应用的 update 用了这么长时间呢？他还给了我一个截图：

```
GapInterfaceRequestLogAspect -cost:46method:execution(MemberController.updateMember(..))req:[{"memberId":"329013069056","mobile":""}
GapInterfaceRequestLogAspect -cost:107method:execution(MemberController.updateMember(..))req:[{"memberId":"000114575434","mobile":""}
GapInterfaceRequestLogAspect -cost:83method:execution(MemberController.updateMember(..))req:[{"memberId":"000013455482","mobile":""}
GapInterfaceRequestLogAspect -cost:111method:execution(MemberController.updateMember(..))req:[{"memberId":"659933502058","mobile":""}
GapInterfaceRequestLogAspect -cost:91method:execution(MemberController.updateMember(..))req:[{"memberId":"341310256761","mobile":""}
GapInterfaceRequestLogAspect -cost:101method:execution(MemberController.updateMember(..))req:[{"memberId":"000116952140","mobile":""}
GapInterfaceRequestLogAspect -cost:76method:execution(MemberController.updateMember(..))req:[{"memberId":"326013262796","mobile":""}
GapInterfaceRequestLogAspect -cost:73method:execution(MemberController.updateMember(..))req:[{"memberId":"001310272534","mobile":""}
GapInterfaceRequestLogAspect -cost:80method:execution(MemberController.updateMember(..))req:[{"memberId":"334608228018","mobile":""}
GapInterfaceRequestLogAspect -cost:39method:execution(MemberController.updateMember(..))req:[{"memberId":"336302372023","mobile":""}
GapInterfaceRequestLogAspect -cost:44method:execution(MemberController.updateMember(..))req:[{"memberId":"003805385470","mobile":""}
GapInterfaceRequestLogAspect -cost:58method:execution(MemberController.updateMember(..))req:[{"memberId":"337707927169","mobile":""}
GapInterfaceRequestLogAspect -cost:62method:execution(MemberController.updateMember(..))req:[{"memberId":"003806175609","mobile":""}
GapInterfaceRequestLogAspect -cost:59method:execution(MemberController.updateMember(..))req:[{"memberId":"005218654583","mobile":""}
GapInterfaceRequestLogAspect -cost:57method:execution(MemberController.updateMember(..))req:[{"memberId":"003804467731","mobile":""}
GapInterfaceRequestLogAspect -cost:75method:execution(MemberController.updateMember(..))req:[{"memberId":"317005699688","mobile":""}
GapInterfaceRequestLogAspect -cost:19method:execution(MemberController.updateMember(..))req:[{"memberId":"004004036349","mobile":""}
GapInterfaceRequestLogAspect -cost:18method:execution(MemberController.updateMember(..))req:[{"memberId":"343612632052","mobile":""}
GapInterfaceRequestLogAspect -cost:11method:execution(MemberController.updateMember(..))req:[{"memberId":"000313166613","mobile":""}
```

从这个图中可以看到时间在 100 毫秒左右。根据我的经验，一个 SQL 执行 100ms，对实时业务来说，确实有点长了。

但是这个时间是长还是短，还不能下结论。要是业务需要必须写成这种耗时的 SQL 呢？

接着他又给我发了 TPS 图。如下所示：



这个 TPS 图确实.....有点乱！还记得前面我对 TPS 的描述吧，在一个场景中，TPS 是要有阶梯的。

如果你在递增的 TPS 场景中发现了问题，然后为了找到这个问题，用同样的 TPS 级别快速加起来压力，这种方式也是可以的。只是这个结果不做为测试报告，而是应该记录到调优报告当中。

而我们现在看到的这个 TPS 趋势，那真是哪哪都挨不上呀。如此混乱的 TPS，那必然是性能有问题。

他还告诉了我两个信息。

1. 有 100 万条参数化数据；
2. GC 正常，dump 文件也没有死锁的问题。

这两个信息应该说只能是信息，并不能起到什么作用。另外，我也不知道他说的“GC 正常”是怎么个正常法，只能相信他说的。

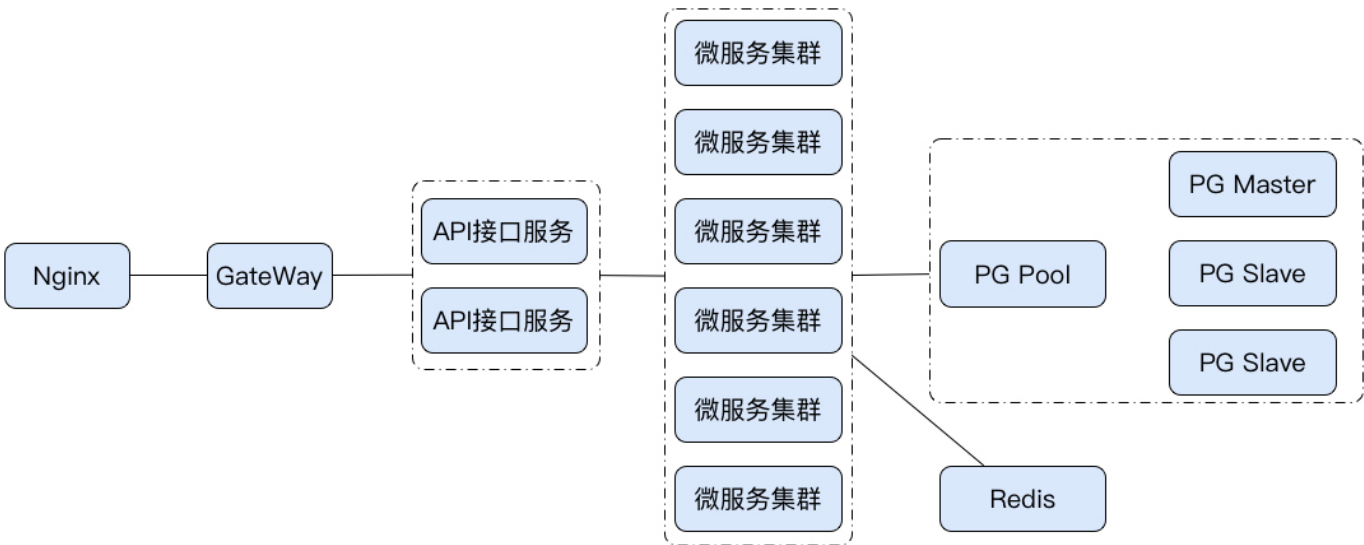
以上就是基本的信息了。

分析过程

照旧，先画个架构图出来看看。

每次做性能分析的时候，我几乎都会先干这个事情。只有看了这个图，我心里才踏实。才能明确知道要面对的系统范围有多大；才能在一个地方出问题的时候，去考虑是不是由其他地方引起的；才能跟着问题找到一条条的分析路径.....

下面是一张简单的架构图，从下面这张架构图中可以看到，这不是个复杂的应用，是个非常典型的微服务结构，只是数据库用了 PostgreSQL 而已。



由于这个问题反馈的是从服务集群日志中看到的 update 慢，所以后面的分析肯定是直接对着数据库去了。

这里要提醒一句，我们看到什么现象，就跟着现象去分析。这是非常正规的思路吧。但就是有一些人，明明看着数据库有问题，非要瞪着眼睛跟应用服务器较劲。

前不久就有一个人问了我一个性能问题，说是在压力过程中，发现数据库 CPU 用完了，应用服务器的 CPU 还有余量，于是加了两个数据库 CPU。但是加完之后，发现数据库 CPU 使用率没有用上去，反而应用服务器的 CPU 用完了。我一听，觉得挺合理的呀，为什么他

在纠结应用服务器用完了呢？于是我就告诉他，别纠结这个，先看时间耗在哪里。结果发现应用的时间都耗在读取数据库上了，只是数据库硬件好了一些而已。

因为这是个在数据库上的问题，所以我直接查了数据库的资源。

```
procs -----memory----- --swap-- -----io---- -system-- -----cpu-----
r  b      swpd      free      buff      cache      si      so      bi      bo      in      cs      us      sy      id      wa      st
1  1          0  1197568      6136  260157664      0      0      0      664      124      0      0      1      0  99      0      0
5  1          0  1203620      6136  260151248      0      0  303624  10815  73773  42879      3      2  92      3      0
1  1          0  1182444      6136  260168736      0      0  300032  15552  78695  45196      3      2  92      3      0
1  1          0  1152212      6144  260194128      0      0  309256   2460  73516  42288      3      2  93      3      0
2  1          0  1197348      6144  260148896      0      0  296968  17292  76506  43484      3      2  92      3      0
2  2          0  1235492      6144  260107904      0      0  311296   2864  79335  45390      3      2  92      3      0
1  2          0  1200340      6144  260141488      0      0  311298   2894  79563  45423      3      2  92      3      0
0  1          0  1138832      7340  260205776      0      0  300974  15666  72627  41078      3      2  92      3      0
3  0          0  1146696      7340  260194848      0      0  309248   2524  73891  42050      3      2  93      3      0
0  3          0  1183324      7340  260152800      0      0  305328  11188  79397  44542      4      3  91      3      0
4  0          0  1206660      7348  260123856      0      0  300884   8454  75849  43136      3      2  92      3      0
1  1          0  1204052      7348  260124736      0      0  313344   2518  72262  41341      3      2  92      3      0
0  1          0  1238568      7348  260090640      0      0  295088  17016  67868  38200      2      2  93      3      0
```

查看 vmstat，从这个结果来看，系统资源确实没用上。不过，请注意，这个 bi 挺高，能达到 30 万以上。那这个值说明了什么呢？我们来算一算。

bi 是指每秒读磁盘的块数。所以要先看一下，一块有多大。

拼课微信：1716143661

复制代码

```
1 [root@7dgroup1 ~]# tune2fs -l /dev/vda1 | grep "Block size"
2 Block size: 4096
3 [root@7dgroup1 ~]#
```

那计算下来大约就是：

$$(300000 * 1024) / 1024 / 1024 \approx 293M$$

将近 300M 的读取，显然这个值是不低的。

接下来查看 I/O。再执行下 iostat 看看。

0.00	1.00	4303.00	807.00	20400.00	2922.00	124.33	39.72	11.01	11.90	11.22	0.19	90.33
%user	%nice	%system	%iowait	%steal	%idle							
3.37	0.00	2.58	3.04	0.00	91.01							
rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
0.00	0.00	0.00	29.00	0.00	132.00	9.10	0.04	1.38	0.00	1.38	0.03	0.10
0.00	0.00	4790.00	363.50	309074.00	4265.50	121.60	54.54	10.68	11.27	2.88	0.18	94.95
%user	%nice	%system	%iowait	%steal	%idle							
1.73	0.00	1.45	2.97	0.00	93.86							
rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	4643.50	445.00	297108.00	16231.25	123.16	58.90	11.54	11.61	10.80	0.19	95.20
%user	%nice	%system	%iowait	%steal	%idle							
1.71	0.00	1.45	3.15	0.00	93.69							
rrqm/s	wrqm/s	r/s	w/s	rkB/s	wkB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	4906.50	217.50	310400.00	2956.00	122.31	56.28	11.04	11.33	4.51	0.19	95.45

从这个结果来看，%util 已经达到了 95% 左右，同时看 rkB/s 那一列，确实在 300M 左右。

接着在 master 上面的执行 iotop。

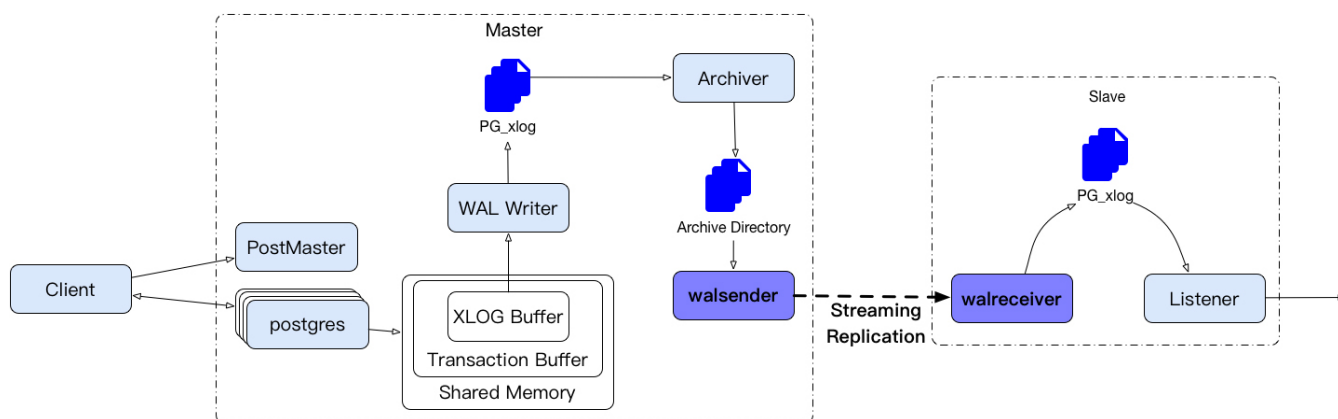
Total DISK READ :		304.81 M/s		Total DISK WRITE :		10.86 M/s	
Actual DISK READ:		304.81 M/s		Actual DISK WRITE:		4.07 M/s	
TID	PRI	USER	DISK READ	DISK WRITE	SWAPIN	IO>	COMMAND
18622	be/4	postgres	304.81 M/s	5.77 M/s	0.00 %	56.07 %	postgres: walsender postgres 172.16.20.68(34952) idle
25556	be/4	postgres	0.00 B/s	185.40 K/s	0.00 %	5.14 %	postgres: postgres gap_new 172.16.10.104(36976) UPDATE
14106	be/4	postgres	0.00 B/s	139.05 K/s	0.00 %	3.82 %	postgres: postgres gap_new 172.16.10.104(44960) COMMIT
9817	be/4	postgres	0.00 B/s	618.01 K/s	0.00 %	2.98 %	postgres: walwriter
16122	be/4	postgres	0.00 B/s	69.53 K/s	0.00 %	2.87 %	postgres: postgres gap_new 172.16.10.104(48444) COMMIT
16116	be/4	postgres	0.00 B/s	84.98 K/s	0.00 %	2.70 %	postgres: postgres gap_new 172.16.10.104(48420) COMMIT
16112	be/4	postgres	0.00 B/s	123.60 K/s	0.00 %	2.15 %	postgres: postgres gap_new 172.16.10.104(48404) COMMIT
16119	be/4	postgres	0.00 B/s	92.70 K/s	0.00 %	1.73 %	postgres: postgres gap_new 172.16.10.104(48432) COMMIT
2867	be/4	postgres	0.00 B/s	77.25 K/s	0.00 %	1.58 %	postgres: postgres gap_new 172.16.10.104(53996) COMMIT
16117	be/4	postgres	0.00 B/s	92.70 K/s	0.00 %	1.56 %	postgres: postgres gap_new 172.16.10.104(48424) COMMIT
16121	be/4	postgres	0.00 B/s	92.70 K/s	0.00 %	1.54 %	postgres: postgres gap_new 172.16.10.104(48440) COMMIT
16133	be/4	postgres	0.00 B/s	146.78 K/s	0.00 %	1.52 %	postgres: postgres gap_new 172.16.10.104(48484) COMMIT
16127	be/4	postgres	0.00 B/s	146.78 K/s	0.00 %	1.48 %	postgres: postgres gap_new 172.16.10.104(48464) idle in transaction
16118	be/4	postgres	0.00 B/s	139.05 K/s	0.00 %	1.32 %	postgres: postgres gap_new 172.16.10.104(48428) COMMIT
16113	be/4	postgres	0.00 B/s	100.43 K/s	0.00 %	1.03 %	postgres: postgres gap_new 172.16.10.104(48408) COMMIT
16124	be/4	postgres	0.00 B/s	69.53 K/s	0.00 %	1.02 %	postgres: postgres gap_new 172.16.10.104(48452) COMMIT
16132	be/4	postgres	0.00 B/s	108.15 K/s	0.00 %	0.91 %	postgres: postgres gap_new 172.16.10.104(48480) COMMIT
16125	be/4	postgres	0.00 B/s	54.08 K/s	0.00 %	0.89 %	postgres: postgres gap_new 172.16.10.104(48456) idle in transaction
1941	be/4	postgres	0.00 B/s	30.90 K/s	0.00 %	0.84 %	postgres: postgres gap_new 172.16.10.104(52518) COMMIT
25559	be/4	postgres	0.00 B/s	54.08 K/s	0.00 %	0.82 %	postgres: postgres gap_new 172.16.10.104(36990) COMMIT
16134	be/4	postgres	0.00 B/s	61.80 K/s	0.00 %	0.76 %	postgres: postgres gap_new 172.16.10.104(48488) COMMIT
16128	be/4	postgres	0.00 B/s	46.35 K/s	0.00 %	0.76 %	postgres: postgres gap_new 172.16.10.104(48468) COMMIT
16126	be/4	postgres	0.00 B/s	23.18 K/s	0.00 %	0.73 %	postgres: postgres gap_new 172.16.10.104(48460) COMMIT
25551	be/4	postgres	0.00 B/s	38.63 K/s	0.00 %	0.70 %	postgres: postgres gap_new 172.16.10.104(36958) COMMIT
15240	be/4	postgres	0.00 B/s	108.15 K/s	0.00 %	0.36 %	postgres: postgres gap_new 172.16.10.104(46814) COMMIT
2860	be/4	postgres	0.00 B/s	84.98 K/s	0.00 %	0.26 %	postgres: postgres gap_new 172.16.10.104(53968) COMMIT
16120	be/4	postgres	0.00 B/s	46.35 K/s	0.00 %	0.22 %	postgres: postgres gap_new 172.16.10.104(48436) COMMIT
16123	be/4	postgres	0.00 B/s	54.08 K/s	0.00 %	0.20 %	postgres: postgres gap_new 172.16.10.104(48448) COMMIT
16115	be/4	postgres	0.00 B/s	61.80 K/s	0.00 %	0.17 %	postgres: postgres gap_new 172.16.10.104(48416) COMMIT

我发现 Walsender Postgres 进程达到了 56.07% 的使用率，也就是说它的读在 300M 左右。但是写的并不多，从图上看只有 5.77M/s。

结合上面几个图，我们后面的优化方向就是：**降低读取，提高写入。**

到这里，我们就得说道说道了。这个 Walsender Postgres 进程是干吗的呢？

我根据理解，画了一个 Walsender 的逻辑图：



从这个图中就可以看得出来，Walsender 和 Walreceiver 实现了 PostgreSQL 的 Master 和 Slave 之间的流式复制。Walsender 取归档目录中的内容（敲黑板了呀！），通过网络发送给 Walreceiver，Walreceiver 接收之后在 slave 上还原和 master 数据库一样的数据。

而现在读取这么高，那我们就把读取降下来。

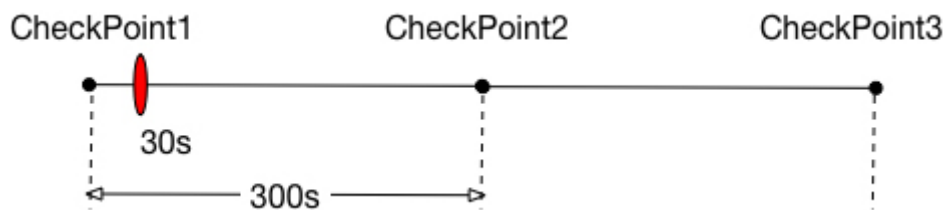
先查看一下几个关键参数：

```
postgres=# show checkpoint_completion_target;
checkpoint_completion_target
-----
0.5
(1 row)
```

```
postgres=# show checkpoint_timeout;
checkpoint_timeout
-----
5min
(1 row)
```

这两个参数对 PostgreSQL 非常重要。checkpoint_completion_target 这个值表示这次 checkpoint 完成的时间占到下一次 checkpoint 之间的时间的百分比。

这样说似乎不太好理解。画图说明一下：



在这个图中 300s 就是 `checkpoint_timeout`，即两次 checkpoint 之间的时间长度。这时若将 `checkpoint_completion_target` 设置为 0.1，那就是说 CheckPoint1 完成时间的目标就是在 30s 以内。


在这样的配置之下，你就会知道 `checkpoint_completion_target` 设置得越短，集中写的内容就越多，I/O 峰值就会高；`checkpoint_completion_target` 设置得越长，写入就不会那么集中。也就是说 `checkpoint_completion_target` 设置得长，会让写 I/O 有缓解。

在我们这个案例中，写并没有多少。所以这个不是什么问题。

但是读取的 I/O 那么大，又是流式传输的，那就是会不断地读文件，为了保证有足够的数
据可以流式输出，这里我把 `shared_buffers` 增加，以便减轻本地 I/O 的压力。

来看一下优化动作：

```
1 checkpoint_completion_target = 0.1
2 checkpoint_timeout = 30min
3 shared_buffers = 20G
4 min_wal_size = 1GB
5 max_wal_size = 4GB
```

 复制代码

其中的 `max_wal_size` 和 `min_wal_size` 官方含义如下所示。

`max_wal_size` (integer):

Maximum size to let the WAL grow to between automatic WAL checkpoints. This is a soft limit; WAL size can exceed `max_wal_size` under special circumstances, like under heavy load, a failing archive_command, or a high `wal_keep_segments` setting. The default is 1 GB. Increasing this parameter can increase the amount

of time needed for crash recovery. This parameter can only be set in the postgresql.conf file or on the server command line.

min_wal_size (integer):

As long as WAL disk usage stays below this setting, old WAL files are always recycled for future use at a checkpoint, rather than removed. This can be used to ensure that enough WAL space is reserved to handle spikes in WAL usage, for example when running large batch jobs. The default is 80 MB. This parameter can only be set in the postgresql.conf file or on the server command line.

请注意，上面的 shared_buffers 是有点过大的，不过我们先验证结果再说。

优化结果

再看 iostat:

```
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.13    0.00    0.42    1.08    0.00   97.36

Device:            rrqm/s   wrqm/s     r/s     w/s    rkB/s   kB/s avgrq-sz avgqu-sz   await  r_await  w_await  svctm  %util
vda                0.00    0.00    0.00    0.00     0.00    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
vdb                0.00    0.00 1159.00   84.00  9480.00 3752.00 21.29  0.48  0.38  0.33  1.12  0.26 32.45

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.13    0.00    0.50    1.07    0.00   97.30

Device:            rrqm/s   wrqm/s     r/s     w/s    rkB/s   kB/s avgrq-sz avgqu-sz   await  r_await  w_await  svctm  %util
vda                0.00    0.00    0.00    0.00     0.00    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
vdb                0.00    0.00 1138.50   85.50  9252.00 3788.00 21.31  0.48  0.39  0.34  1.07  0.27 33.15

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.21    0.00    0.44    1.10    0.00   97.25

Device:            rrqm/s   wrqm/s     r/s     w/s    rkB/s   kB/s avgrq-sz avgqu-sz   await  r_await  w_await  svctm  %util
vda                0.00    0.00    0.00    0.00     0.00    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
vdb                0.00    0.50 1169.00   90.50  9584.00 3944.00 21.48  0.47  0.38  0.34  0.91  0.26 33.30

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.08    0.00    0.53    1.07    0.00   97.32

Device:            rrqm/s   wrqm/s     r/s     w/s    rkB/s   kB/s avgrq-sz avgqu-sz   await  r_await  w_await  svctm  %util
vda                0.00    0.00    0.00    0.00     0.00    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
vdb                0.00    0.50 1137.00  105.50  9304.00 3355.75 20.38  0.47  0.38  0.34  0.81  0.24 30.15

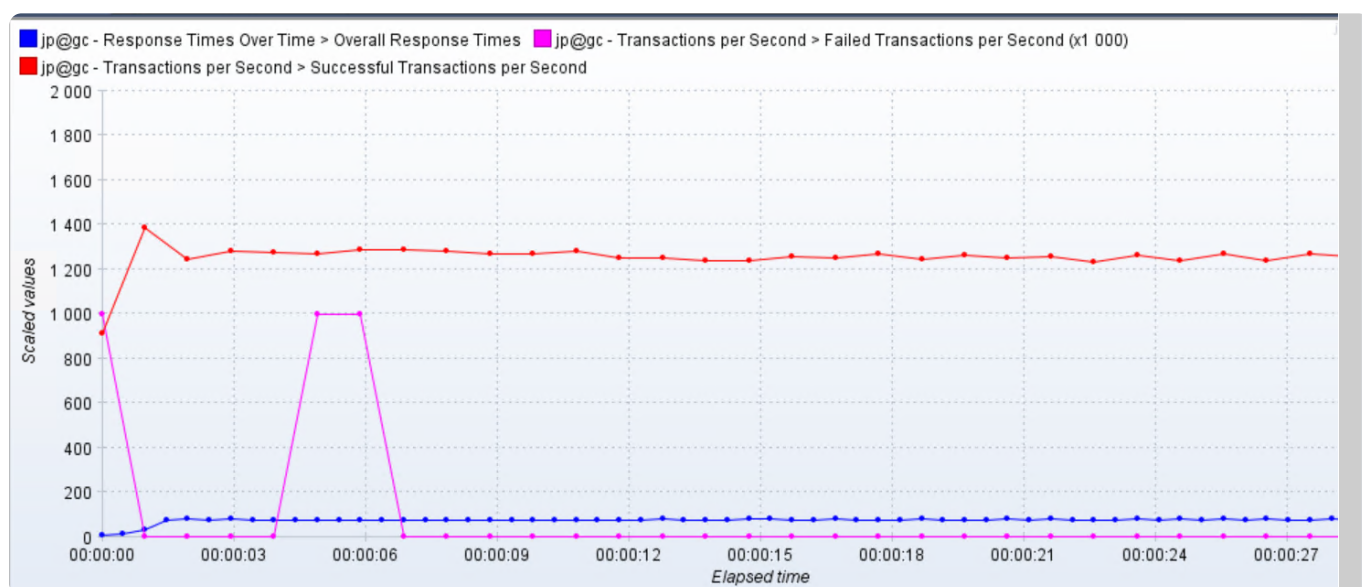
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.15    0.00    0.49    1.04    0.00   97.33

Device:            rrqm/s   wrqm/s     r/s     w/s    rkB/s   kB/s avgrq-sz avgqu-sz   await  r_await  w_await  svctm  %util
vda                0.00    0.00    0.00    0.00     0.00    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
vdb                0.00    6.00 1130.50  109.50  9284.00 3554.25 20.71  0.49  0.39  0.32  1.11  0.24 30.20

avg-cpu:  %user   %nice %system %iowait  %steal   %idle
           1.21    0.00    0.50    1.02    0.00   97.27

Device:            rrqm/s   wrqm/s     r/s     w/s    rkB/s   kB/s avgrq-sz avgqu-sz   await  r_await  w_await  svctm  %util
vda                0.00    0.00    0.00    0.00     0.00    0.00  0.00  0.00  0.00  0.00  0.00  0.00  0.00
vdb                0.00    0.50 1103.00   89.50  9052.00 3100.00 20.38  0.45  0.38  0.33  0.93  0.26 31.05
```

看起来持续的读降低了不少。效果是有的，方向没错。再来看看 TPS:



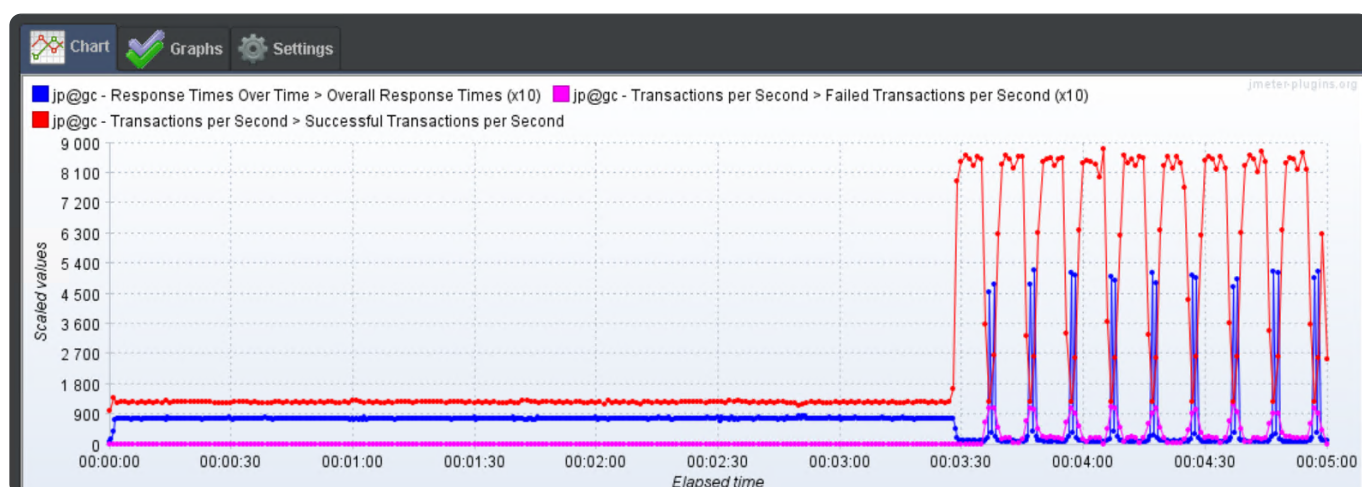
看这里 TPS 确实稳定了很多，效果也比较明显。

这也就达到我们优化的目标了。就像在前面文章中所说的，在优化的过程中，当你碰到 TPS 非常不规则时，请记住，一定要先把 TPS 调稳定，不要指望在一个混乱的 TPS 曲线下做优化，那将使你无的放矢。

问题又来了？

在解决了上一个问题之后，没过多久，另一个问题又抛到我面前了，这是另一个接口，因为是在同一个项目上，所以对问问题的人来说，疑惑还是数据库有问题。

来看一下 TPS：



这个问题很明显，那就是后面的成功事务数怎么能达到 8000 以上？如果让你蒙的话，你觉得会是什么原因呢？

在这里，告诉你我对 TPS 趋势的判断逻辑，那就是 **TPS 不能出现意外的趋势**。

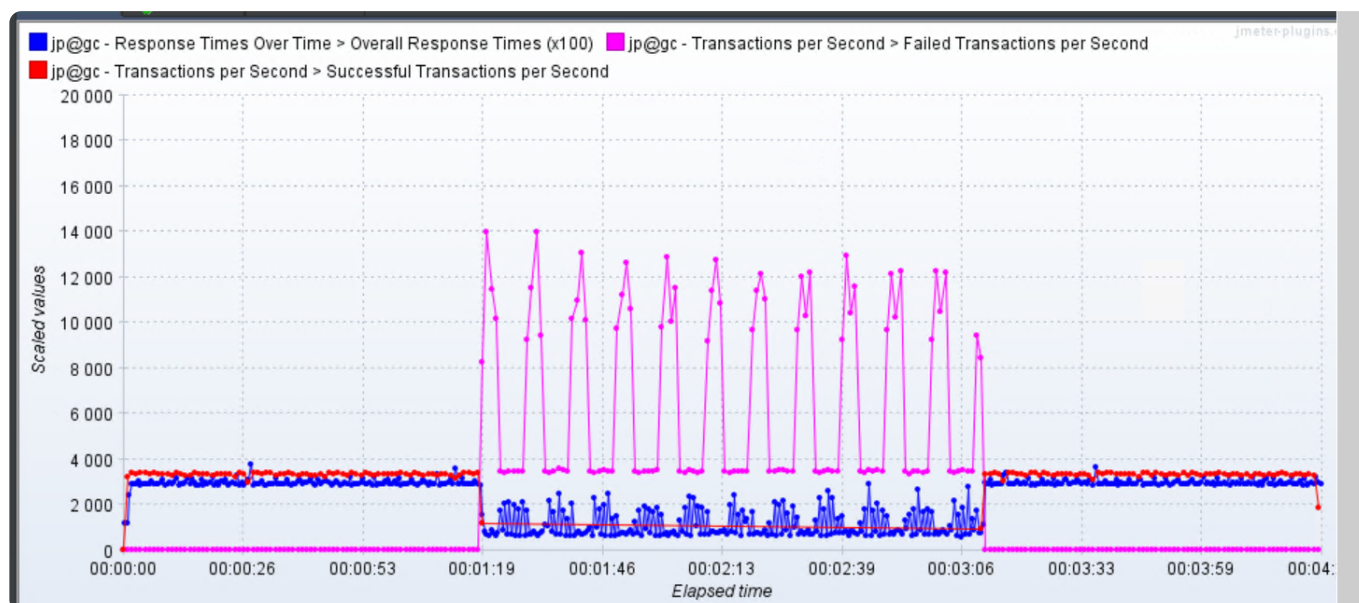
什么叫意外的趋势？就是当在运行一个场景之前就已经考虑到了这个 TPS 趋势应该是个什么样子（做尝试的场景除外），当拿到运行的结果之后，TPS 趋势要和预期一致。

如果没有预期，就不具有分析 TPS 的能力了，最多也就是压出曲线，但不会懂曲线的含义。

像上面的这处 TPS 图，显然就出现意外了，并且是如此大的意外。前面只有 1300 左右的 TPS，后面怎么可能跑到 8000 以上，还全是对的呢？

所以我看到这个图之后，就问了一下：是不是没加断言？

然后他查了一下，果然没有加断言。于是重跑场景。得到如下结果：



从这个图上可以看到，加了断言之后，错误的事务都正常暴露出来了。像这种后台处理了异常并返回了信息的时候，前端会收到正常的 HTTP Code，所以才会出现这样的问题。

这也是为什么通常我们都要加断言来判断业务是否正常。

总结

在性能分析的道路上，我们会遇到各种杂七杂八的问题。很多时候，我们都期待着性能测试中的分析像破案一样，并且最好可以破一个惊天地泣鬼神的大案，以扬名四海。

然而分析到了根本原因之后，你会发现优化的部分是如此简单。

其实对于 PostgreSQL 数据库来说，像 buffer、log、replication 等内容，都是非常重要的分析点，在做项目之前，我建议先把这样的参数给收拾一遍，不要让参数配置成为性能问题，否则得不偿失。

思考题

最后问你两个问题吧。为什么加大 buffer 可以减少磁盘 I/O 的压力？为什么说 TPS 趋势要在预期之内？

欢迎你在评论区写下你的思考，我会和你一起交流。也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

课程学习计划

关注极客时间服务号 每日学习签到

月领 25+ 极客币

【点击】保存图片，打开【微信】扫码>>>



© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 31 | 案例：当磁盘参数导致I/O高的时候，应该怎么办？

下一篇 结束语 | 见过林林总总的乱象，才知未来的无限可能



罗辑思维

2020-03-05

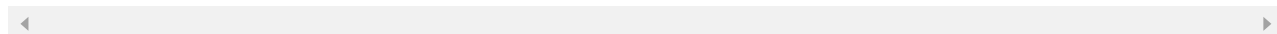
分享下自己学习体会：

为什么缓存可以加速I/O的访问速度？

老师说的缓存应该有两个：操作系统的缓存和PostgreSQL的缓存。它俩作用都是为了把经常访问的数据（也就是热点数据），提前读入到内存中。这样，下次访问时就可以直接从内存读取数据，而不需要经过硬盘，从而加速I/O 的访问速度。...

展开 ▾

作者回复: 认真的同学，必须赞！



1



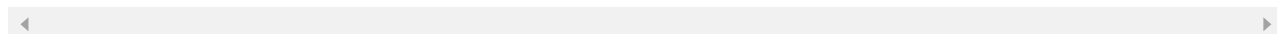
月亮和六便士

2020-03-04

老师，1jmeter tps是150 2jmeter tps是200能说明什么？ 在场景对比中增加jmeter数量我怎么觉得是压力不够呢，怎么能说明server哪个节点有瓶颈

展开 ▾

作者回复: 你觉得压力不够就再加压力看tps能不能增加。



Geek_65c0a2

2020-03-04

刚开始分析时使用vmstat，发现bi高。

如果这时看top命令的话，iowait应该也高吧？

作者回复: 是的，iowait在top中也可以看到，只是在这个例子中没有用top这个命令分析而已。

