MySQL 服务器的 linux 性能优化和扩展技巧

作者： Yoshinori Matsunbu

作者现在是 DeNA 公司的数据库和基础设施架构师。之前在 SUN 公司工作。他也是 HandlerSocket 的作者。这个是 MySQL 的 NoSQL 插件。

本文是根据他的 PPT 整理而成的，如有不正确敬请指教。

本文主要的内容有如下：

1. 内存和 SWAP 空间管理
2. 同步 I/O，文件系统和 I/O 调度
3. 有用的命令和工具：iostat, mpstat, oprofile, SystemTap, gdb

第一部分：内存和 SWAP 空间管理

内存也就是随机访问内存

内存是最终要的硬件部件对于 RDBMS(relation database management system)。

内存的访问速度远远超过 HDD（普通硬盘）/SSD（固态硬盘）

内存： 60ns， 但是还没达到每秒 10W

HDD： 5ms

SSD： 100-500us

他们之间的关系为：

1s = 1000ms

1ms = 1000us

1us = 1000ns

所以 16GB-64GB 对于现在是非常合适的。（好像之前在人人的时候都是 72G）

热点应用的数据都需要缓存在内存中

当然最小化热点数据大小也是很重要的，主要有以下几种措施：

使用紧凑长度的数据类型(SMALLINT 来替代 VARCHAR/BIGINT, TIMESTAMP 来替代 DATETIME 等等)

不要创建无用的索引

删除不必要的数据或者将这些数据移到存档表中，来保证热点的表尽量的小

下面这个测试就是针对不同内存大小服务器的一个测试，测试数据在 20-25GB（200 个数据仓库，运行一小时），使用的是 DBT-2 测试，这是一种密集写的测试，服务器的配置为 Nehalem 2.93 * 8 cores, MySQL 5.5.2, 4 RAID 1+0 HDDs

DBT-2 (W200)	Transactions per Minute	%user	%iowait
Buffer pool 1G	1125.44	2%	30%
Buffer pool 2G	1863.19	3%	28%
Buffer pool 5G	4385.18	5.5%	33%
Buffer pool 30G (All data in cache)	36784.76	36%	8%

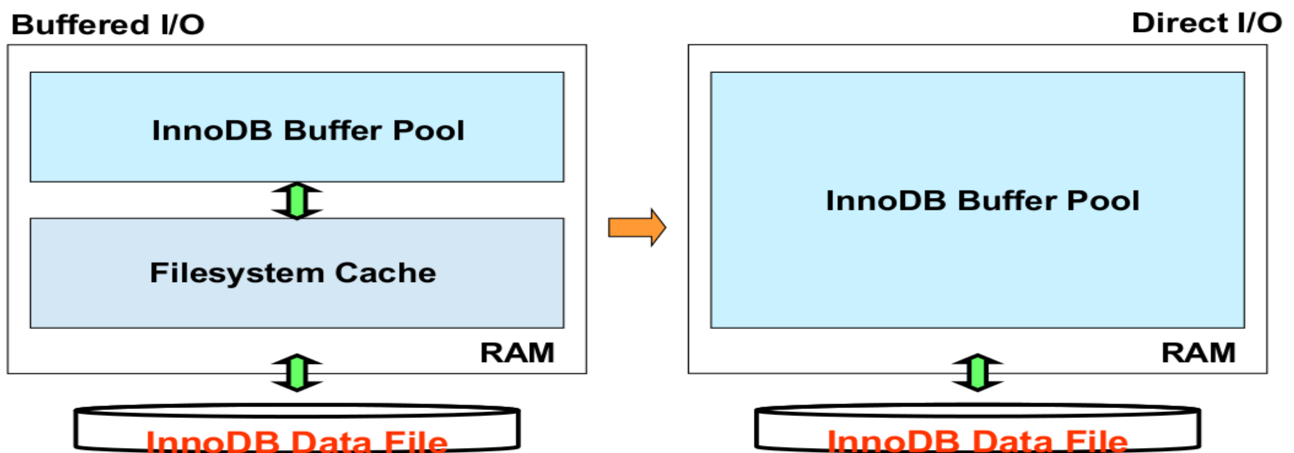
从上面这个表格中我们可以很明显看到巨大的差异当数据全部缓存到内存中。

内存大小会影响所有操作，不管是 SELECT，还是 INSERT/UPDATE/DELETE 操作。

INSERT：当往一个随机排序的索引中插入数据的时候会造成随机的读/写

UPDATE/DELETE： 当更改数据的时候会导致磁盘的读/写

还有一个提高性能的方法是使用直接 I/O（Direct I/O）



从上图中我们可以看到 Direct I/O 就是直接跳过了文件系统的 cache。

Direct I/O 对于完全利用内存是非常重要的。我们可以通过设置 innodb_flush_method=O_DIRECT 来运行。

注：文件 I/O 必须是 512byte 为一个单位，同时 O_DIRECT 不能用在 InnoDB 日志文件，二进制日志文件，MyISAM 引擎，PostgreSQL 数据文件等等。

不要分配太多的内存

这个其实只要分配到足够其它应用程序使用，而不要最后导致系统没有内存可用。

```
user$ top
Mem:  32967008k total, 32808696k used,  158312k free,   10240k buffers
Swap: 35650896k total, 4749460k used, 30901436k free,  819840k cached
 PID USER     PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
5231 mysql    25   0 35.0g  30g  324 S  0.0 71.8   7:46.50 mysqld
```

上图中我们可以看到总共系统 32G 内存，而 Mysqld 已经使用了 30G，而系统居然还只有 150M 可用，这样是非常危险。

当系统没有内存可用时会发生什么事情呢？

减少文件系统缓存来分配内存空间，这个文件系统缓存就是上图中 cached 部分

替换掉一些进行来分配内存空间。也就是将一些内存空间移动到 SWAP

SWAP 是坏的

进程空间会写入到磁盘上（swap out），而这些进程空间本应该是写入到内存中的。

当访问磁盘上的进程空间会导致磁盘读写（swap in）

同时会产生巨量的随机磁盘读写

那也许有些人会想到把 swap 大小设置为 0，但是这样其实是非常危险的。

因为当内存和 SWAP 都不可用的时候的，OOM Killer（out of memory）就会被启用。OOM Killer 会杀掉任何进程来分配内存空间。

最耗费内存的进程会被最先杀掉，在 mysql 服务器上这个一般是 mysqld 进程

mysqld 会被中止关闭，而在重启时候会进行崩溃修复。

OOM Killer 的策略是根据/proc/<pid>/oom_score 来进行倒序排列，也就是 oom_score 最大的会被第一个干掉

通常 mysqld 会拥有最高的值，因为 oom_score 是根据内存大小，CPU 时间，运行时间来判断。

OOM Killer 杀死进程会花费很长的时间，而这期间我们不能干任何事情。

所以不要设置 swap 为 0

```
top - 01:01:29 up  5:53,  3 users,  load average: 0.66, 0.17, 0.06
Tasks: 170 total,   3 running, 167 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us, 24.9%sy,  0.0%ni,75.0%id,0.2%wa,0.0%hi, 0.0%si,0.0%st
Mem:  32967008k total, 32815800k used,   151208k free,     8448k buffers
Swap:         0k total,         0k used,        0k free,   376880k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
26988 mysql     25   0   30g  30g 1452 R 98.5 97.7   0:42.18 mysqld
```

上图中我们看到 swap 被设置为了 0，而一旦没有内存可用 OOM Killer 就会被启用。

一些 CPU 核心会耗尽 100%的系统资源。在上图中我们就看到的就是一个 CPU 核使用 100%的 CPU 资源。而这个时候连接终端(SSH)就会断掉。

所以 swap 是不好的，但是 OOM Killer 更不好。

如果/proc/<PID>/oom_adj 被设置为-17，OOM Killer 就不会杀掉这个进程。所以给 SSHD 进程设置为-17 是一个有效防止断线的方法。

echo -17 > /proc/<PID>/oom_adj

但是不要给 mysqld 设置为-17，因为如果最耗内存的进程没被杀死，linux 依然没有任何可用的内存。而我们就会在很长很长很长的时间内没法干任何事情。

因此，对于一个生产环境的系统 SWAP 是必须的。但是我们同样不希望 Mysql 进行 swap out。

我们就需要知道 mysql 中哪些东西耗费内存

RDBMS：主要的进程空间是被使用的（innodb_buffer_pool, key_buffer, sort_buffer 等等），有时候文件系统的 cache 也会被使用(MyISAM 引擎的文件等等)

管理操作：（备份等等），这个时候主要是文件系统 cache 会被使用

我们要让 mysql 在内存中，也不要分配大量的文件系统 cache。

要特别注意备份操作

因为在备份的时候往往会拷贝大文件，而拷贝大文件就会使用到 swap

```
Mem:  32967008k total, 28947472k used,  4019536k free,   152520k buffers
Swap: 35650896k total,        0k used, 35650896k free,   197824k cached
PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 5231 mysql     25   0 27.0g  27g  288 S  0.0 92.6   7:40.88 mysqld
```

Copying 8GB datafile

```
Mem:  32967008k total, 32808696k used,   158312k free,    10240k buffers
Swap: 35650896k total, 4749460k used, 30901436k free, 8819840k cached
 PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 5231 mysql      25   0 27.0g  22g  324 S  0.0 71.8   7:46.50 mysqld
```

这个时候我们可以设置/etc/sysctl.conf 中 vm.swappiness=o 来避免这个，而默认值是 60

我们看看下图就知道前后的区别了

```
Mem:  32967008k total, 28947472k used,  4019536k free,   152520k buffers
Swap: 35650896k total,        0k used, 35650896k free,   197824k cached
PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 5231 mysql     25   0 27.0g  27g  288 S  0.0 91.3   7:55.88 mysqld
```

Copying 8GB of datafile

```
Mem:  32967008k total, 32783668k used,   183340k free,     3940k buffers
Swap: 35650896k total,      216k used, 35650680k free,  4117432k cached
PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 5231 mysql     25   0 27.0g  27g  288 S  0.0 80.6   8:01.44 mysqld
```

我们看到，同样是拷贝大文件，下面这个 swap 才之用了 216K

这是因为当物理内存耗尽的时候，linux 内核会减少文件系统 cache 作为最高优先级（低优先级就增加那个值）

当文件系统 cache 也没有可用的时候，就会开始使用 swap。而这个 OOM Killer 也不会被启用，因为还有足够的 swap 空间呢。这样才是安全的。

内存分配

mysqld 使用 malloc()/mmap()方法来进行内存分配

如果要使用更快更多并发的内存就要用 tcmalloc()这样的方法

安装 Google Perftools(tcmalloc 被包含在了里面)

```
yum install libunwind
cd google-perftools-1.5 ; ./configure --enable-frame-pointers; make; make install
export LD_PRELOAD=/usr/local/lib/tcmalloc_minimal.so;
mysqld_safe &
```

而对于 InnoDB 来说它会使用它自己的内存分配器

这个可以在 InnoDB Plugin 中进行更改

如果 innodb_use_sys_malloc=1(默认为 1)，InnoDB 就会使用操作系统的内存分配器

这样 tcmalloc 通过设置 LD_PRELOAD 就会被使用。

下面这个是对 2 种不同的内存分配器进行测试，从中可以看到在内存越大时候，这个差距也越明显。平台还是 Nehalem 2.93 * 8 cores, MySQL 5.5.2, 数据量也是 20-25GB（200 个仓库运行 1 个小时）

	Default allocator	tcmalloc_minimal	%user	up
Buffer pool 1G	1125.44	1131.04	2%	+0.50%
Buffer pool 2G	1863.19	1881.47	3%	+0.98%
Buffer pool 5G	4385.18	4460.50	5.5%	+1.2%
Buffer pool 30G	36784.76	38400.30	36%	+4.4%

要个别注意分配给每个 session 的内存

不要分配超过需求过多的的内存大小（特别是针对每个 session 的内存）

因为分配 2MB 内存比分配 128KB 内存会花更多的时间。当分配内存小于等于 512MBLinux malloc()方法内部会调用 brk()方法，其它时候会调用 mmap()。

在一些情况下，分配给每个 session 过多的内存回到反向的性能影响。

SELECT * FROM huge_myisam_table LIMIT 1;

SET read_buffer_size = 256*1024; (256KB)

- -> 0.68 second to run 10,000 times

SET read_buffer_size = 2048*1024; (2MB)

- -> 18.81 seconds to run 10,000 times

从上面我们可以很明显的看到差距。

在大部分情况都不要分配超过需要过多的内存，当然也有特别的场景(比如：MyISAM + LIMIT + FullScan)

第二部分：同步 I/O，文件系统和 I/O 调度

文件 i/O 和同步写

RDBMS 会经常调用 fsync()方法（每一次事务提交，检查点等等）

确认使用 RAID 卡上的电池备份写缓存（BBWC Battery Backed up Write Cache）

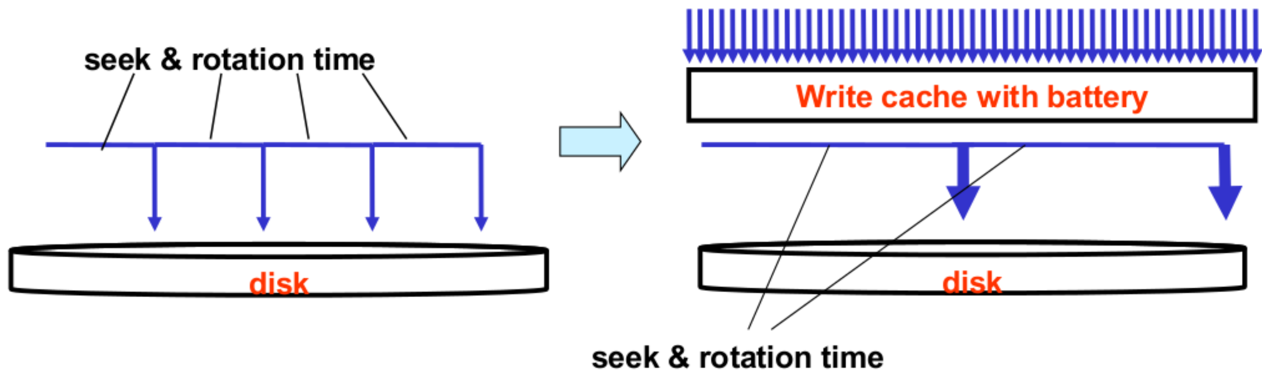
10000+次的 fsync()每秒，而不用 BBWC 会减少 200 次左右。这个都是在磁盘的情况下。

为了安全的原因需要关闭写缓存。

不要在文件系统中设置“写隔离”（在很多情况下默认都是打开的）

就算使用了 BBWC，写入到磁盘还是会非常慢。这是因为一旦打开写隔离，那只有把所有数据写入到磁盘才会关闭隔离。

Ext3 中通过 mount -o barrier=0,在 xfs 中是 mount -o nobarrier,而在 drbd 中是在 drbd.conf 文件中写入 no-disk-barrier。



写隔离技术对于防止脏页是非常有作用的，但是在 mysql 服务器上我们可以关闭，因为都是内部通过事务来提交了。对于其它应用的服务器我们要审慎对待。

复写还是追加写

一些文件是复写的（固定文件大小的），其它的是追加写的（增长的文件长度的）

复写： InnoDB 日志文件

追加写： 二进制日志文件

追加写+fsync()比复写+fsync()要慢的多，这是因为追加写每次都要分配文件需要的空间，同时元数据需要通过每个 fsync()来刷新到磁盘上。

对于复写可以达到 10000+每秒的 fsync，而追加写只有 3000 次左右。

追加写的速度依赖于文件系统。

copy-on-write 的文件系统如 Solaris 的 ZFS 就会对于追加写足够快，可以达到 7000+次。

特别小心设置 sync-binlog=1 为二进制日志，设置为 1 的时候会每个事务写入一次就会自动同步硬盘一次。这样效率会非常差

这个时候可以考虑 ZFS

检查“预分配二进制日志”的工作日志。Http://forge.mysql.com/worklog/task.php?id=4925

不要太频繁的更新文件

innodb_autoextend_increment=20(默认为 8)，这个表示表空间文件每次扩展空间都到 20M

快速文件 I/O 健康检测

启用 BBWC，并且写隔离是关闭的。

复写+fsync()测试：运行 mysqlslap 插入（InnoDB， 单线程， innodb_flush_log_at_trx_commit=1 log buffer 每次事务提交都会写入 log file，并且将数据刷新到磁盘中去）；检查的 qps 超过了 1000.

```
$ mysqlslap --concurrency=1 --iterations=1
--engine=innodb \
  --auto-generate-sql --auto-generate-sql-load-
type=write \
  --number-of-queries=100000
```

具体使用方法可以参考 http://dev.mysql.com/doc/refman/5.1/en/mysqlslap.html

缓冲区和异步写

一些文件 I/O 操作既不是使用 Direct I/O，也不是使用同步写，如：文件复制，MyISAM, mysqldump, innodb_flush_log_at_trx_commit=2 等等

在文件系统缓存的脏页最终都要被刷新到磁盘上去。pdflush 用作刷新到磁盘上的，它最大可以 8 个线程同时进行。

这个是高度依赖于 vm.dirty_background_ratio 和 vm.dirty_ratio 这 2 个系统参数的。当脏页数量达到 dirty_background_ratio（默认是 10%，64GB 内存的话就是当 cache 达到 6.4GB）的时候就会开始刷新到磁盘上。

当达到 dirty_ratio 的时候就会强制进行刷新到磁盘，默认是 40%

强制和粗鲁的脏页刷新是有问题的。当大幅增加传输延迟时间，会导致所有的 buffer 的写操作都变成了同步的。

过分的刷新脏页到磁盘

- 执行刷新，会产生大量的写操作
- 减少 vm.dirty_background_ratio 的值
- 升级内核到 2.6.22 或者更高版本
  - pdflush 线程会给每个设备进行分配，刷新一个慢设备的时候不会阻碍其它设备的 pdflush 线程。

文件系统—EXT3

这是一种现在最广泛使用的文件系统，但是它明显不是最好的。

首先它在删除大文件的会花费很长的时间：在执行的时候内部会有很多随机的磁盘 I/O（HDD）。而对于 mysql 来说，当执行 DROP table 的时候，所有 open/lock 表的客户端线程都会被 block 掉(LOCK_open mutex)。还有要特别注意使用 MyISAM, 使用 innodb_file_per_table 的 InnoDB，以及 PBXT 引擎等。

写文件是串行化的

- 串行化是通过 i-mutex(互斥)，分配给每个 inode
- 有时候它比分配单个大文件会快。
- 对于快速的存储设备缺少优化（如 PCI-E 接口的 SSD）

使用“dir_index”来加快搜索文件，这个需要在文件系统中增加这个属性，tune2fs -O +dir_index /dev/hda5

关闭 barrier。

文件系统—XFS/EXT2/BTRFS

xfs 的特点

- 快速删除文件
- 当使用 O_DIRECT 可以进行并发写入到一个文件
- 在 RHEL 中没有官方支持
- 可以设置“nobarrier”来关闭写隔离

ext2

- 更快速的写，因为它不支持日志，所以出现问题不能进行恢复
- fsck 的时间很长
- 在 active-active 的冗余环境下使用（比如 MySQL 的 replication）
- 在一些情况下，ext2 拥有更好的性能

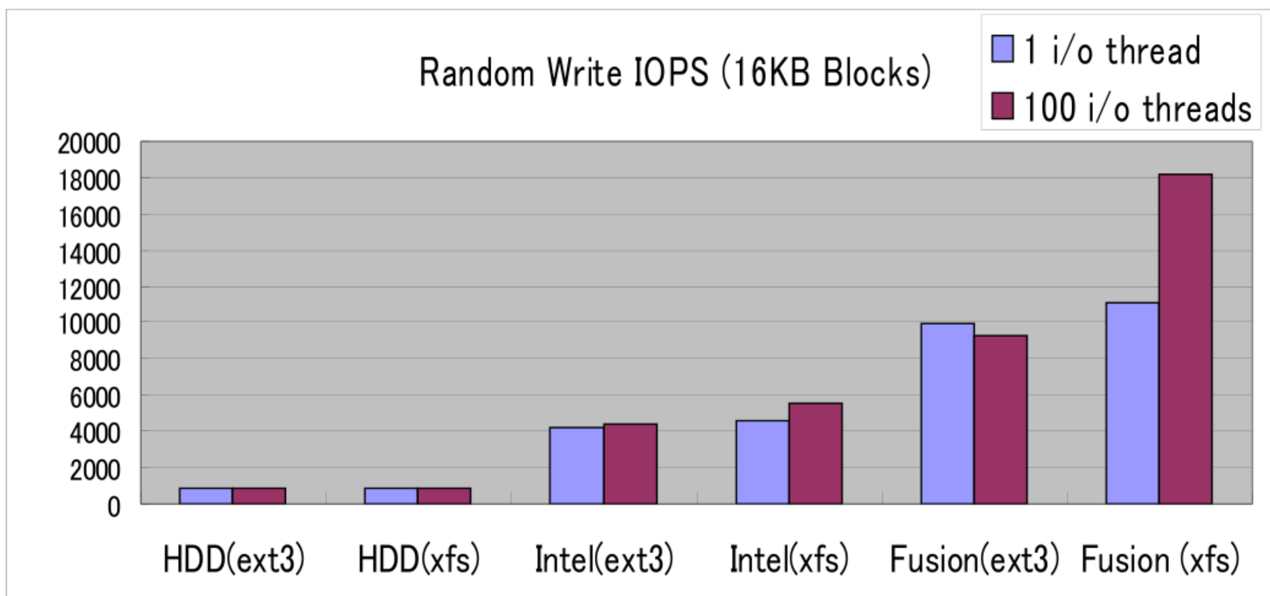
btrfs（开发中）

这是一种跟 ZFS 一样的 copy-on-write 的文件系统

支持事务（没有 half-block 更新）

snapshot 备份无需额外的开销

下图就是 ext3 和 xfs 在不同的磁盘上的随机写的一个对比图。HDD 就是普通磁盘，Intel 应该是普通的 SATA 接口的 SSD，而 FUSION 应该是 pci-e 接口的 SSD



上面的 HDD 是 4 块 SAS RAID1。

I/O 调度器

注：RDBMS（特别是 InnoDB）都会调度 I/O 请求，所以理论上 Linux I/O 调度器并不是必须的。

Linux 的 I/O 调度器可以有效的控制 I/O 请求，I/O 调度器类型和队列大小都是要考虑的问题。

Linux I/O 调度器的类型（根据 RHEL5，内核 2.6.10）

noop: 排序进入的 I/O 请求通过逻辑 block 地址，其实就是 FIFO，先进先出。

Deadline:读请求(sync)的请求比写请求(async)拥有更高的优先级。其它的就是 FIFO，这样就能避免 I/O 请求饥饿的问题。

cfg(默认)：对于每个 I/O 线程公平的策略 I/O，它会对所有的 I/O 请求进行逻辑 block 地址重新进行排序，这样减少了查找 block 地址的时间。

Anticipatory: 在 2.6.33 内核中已经删除，所以请不要再进行使用了。

下面会并发运行 2 个压力测试程序

多线程的随机磁盘读（默认 RDBMS 读）

单线程的复写+fsync()（模拟 redo 日志写）

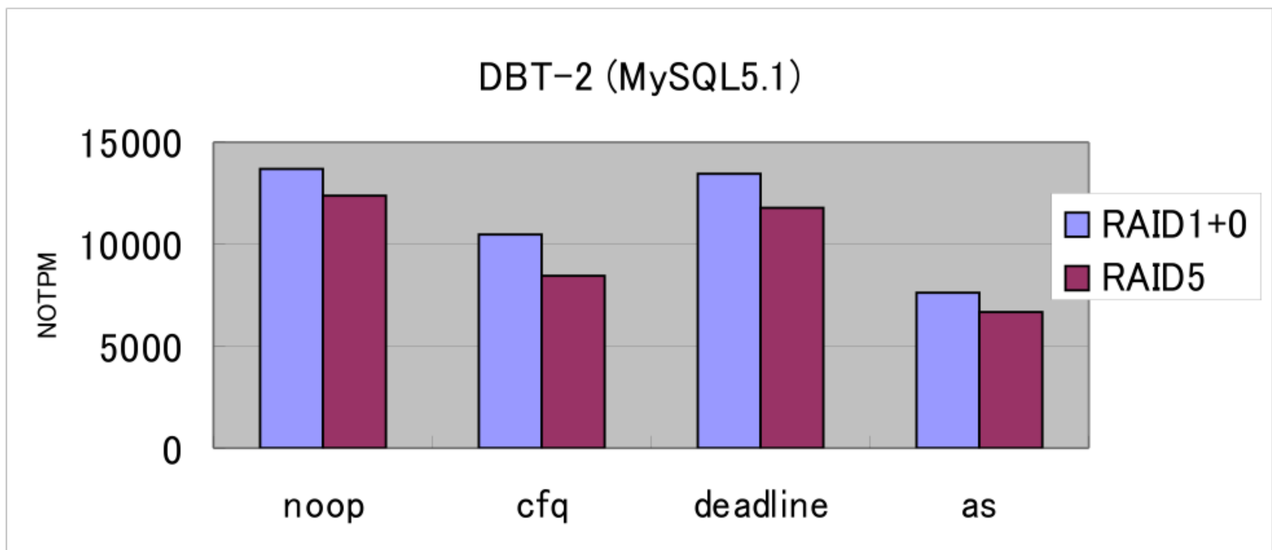
Random Read i/o threads	write+fsync() running	Scheduler	reads/sec from iostat	writes/sec from iostat
1	No	noop/deadline	260	0
		cfq	260	0
100	No	noop/deadline	2100	0
		cfq	2100	0
1	Yes	noop/deadline	212	14480
		cfq	248	246
		noop/deadline	1915	12084
		cfq	2084	0

从上面图中我们可以很容易的看到 cfq 和 noop 的差距。操作为 RHEL5.3 和 SUSE11，4 HDD 的 RAID 1+0。

在 RDBMS 中，写的 IOPS 通常都非常高，因为 HDD 写 cache 每秒需要控制成千上万的事务提交(write+fsync)

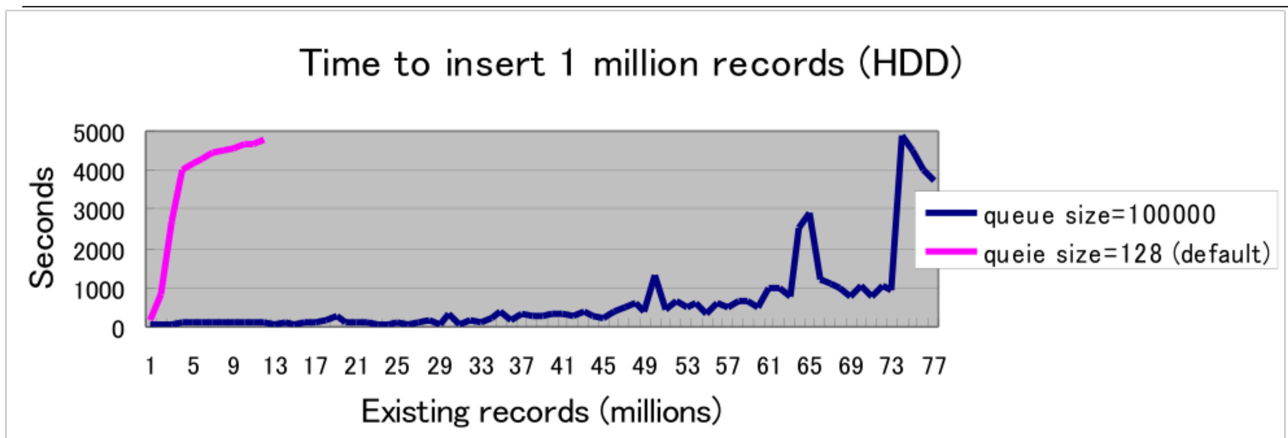
写入的 IOPS 会被调整为每个线程读 IOPS，所以很明显的减少总的 IOPS。

下面这个是 4 个不同的 I/O 策略的测试图，使用的 DBT-2 测试，引擎为 InnoDB



可以看到 noop 和 deadline 差距还是很少的，但是比 cfq 还是高出 30%的样子。

下面这个图是更改了 I/O 策略的队列大小后的对比图，所以用的 MyISAM 引擎的比较结果



queue size=N, I/O 调度器就会排序 N 个请求来优化磁盘查找。

MyISAM 引擎不会在内部优化 I/O 请求，它高度依赖 OS 和存储。当对索引进行插入会导致巨量的随机磁盘读写。

增加 I/O 队列大小可以减少磁盘查找的开销。Echo 100000 > /sys/block/sdX/queue/nr_requests

这种操作对于 InnoDB 没有影响，InnoDB 会在内部进行排序 I/O 请求。

有用的命令和工具

iostat

mpstat

oprofile

SystemTap(stap)

gdp

作者讲了这 5 种命令和工具，但是我这边只说到前面 3 个命令和工具。

iostat

每个设备的详细的 I/O 统计数据，对于 RDBMS 非常重要，因为它经常成为 I/O 瓶颈。

Iostat -xm 10 每 10 秒执行一次。主要注意 r/s 和 w/s，svctm 是平均服务时间（milliseconds），而 util 就是（r/s+w/s）*svctm

```
# iostat -xm 10
avg-cpu: %user %nice %system %iowait %steal %idle
         21.16  0.00    6.14   29.77   0.00 42.93
Device: rqm/s wrqm/s    r/s   w/s rMB/s wMB/s avgrq-sz avgqu-sz await svctm %util
   sdb   2.60 389.01 283.12 47.35  4.86  2.19    43.67     4.89 14.76  3.02 99.83
```

(283.12+47.35) * 3.02(ms)/1000 = 0.9980 = 100% util

```
# iostat -xm 10
avg-cpu: %user %nice %system %iowait %steal %idle
         40.03  0.00   16.51   16.52   0.00 26.94
Device: rrqm/s wrqm/s    r/s    w/s rMB/s wMB/s avgrq-sz avgqu-sz await svctm %util
   sdb    6.39 368.53 543.06 490.41  6.71  3.90    21.02     3.29  3.20  0.90 92.66
```

(543.06+490.41) * 0.90(ms)/1000 = 0.9301 = 93% util

svctm 越低意味着 r/s 和 w/s 越高。所以我们不要太相信 util，我们主要关注的是 r/s,w/s 和 svctm 这几个值。如果你的 IOPS 是 1000，那如果 svctm 是 1ms 的话，那 util 就是 100。所以当 svctm 大于 1 的话就算是有问题了。

Mpstat

以前我一直用 vmstat,但是 vmstat 是显示的所有 CPU 的一个平均数，后来发现 mpstat 是能显示每个 CPU 核的统计。经常会发现某个 CPU 核会占满 100%的资源，而其它 CPU 核却还空闲着。而如果你使用 vmstat/top/iostat/sar 你却无法发现难个 CPU 的瓶颈。

你也可以用 mpstat 来检查网络的瓶颈。

```
# vmstat 1
...
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  1 2096472 1645132  18648  19292    0    0  4848     0 1223  517  0  0 88 12  0
 0  1 2096472 1645132  18648  19292    0    0  4176     0 1287  623  0  0 87 12  0
 0  1 2096472 1645132  18648  19292    0    0  4320     0 1202  470  0  0 88 12  0
 0  1 2096472 1645132  18648  19292    0    0  3872     0 1289  627  0  0 87 12  0
```

```
# mpstat -P ALL 1
...
11:04:37 AM  CPU   %user   %nice    %sys %iowait    %irq   %soft  %steal   %idle    intr/s
11:04:38 AM  all    0.00    0.00    0.12   12.33    0.00    0.00    0.00   87.55   1201.98
11:04:38 AM    0    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00    990.10
11:04:38 AM    1    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00      0.00
11:04:38 AM    2    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00      0.00
11:04:38 AM    3    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00      0.00
11:04:38 AM    4    0.99    0.00    0.99   98.02    0.00    0.00    0.00    0.00    206.93
11:04:38 AM    5    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00      0.00
11:04:38 AM    6    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00      4.95
11:04:38 AM    7    0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00      0.00
```

从上面 VMSTAT 的图中我们看 CPU 的空闲度达到了 88%，但是通过 MPSTAT 图中发现是一个 CPU 满了，而其它 CPU 都完全空闲了，这个就是 CPU 资源分配不均。这个在之前我们 nginx cache 服务器上也发现了类似的问题，最终解决后发现性能提升了 30%以上。

Oprofile

oprofile 是可以查看运行进程的 CPU 使用状况的概括。你可以很容易的确认那些方法用掉了这些 CPU 资源。这个工具同时支持系统空间和用户空间。这个工具主要是用于数据库的内部开发者。如果发现有特别的方法占用了大部分的资源，程序员最好跳过这些方法的调用。对于检查低 cpu 活动，IO 限制和互斥等情况没有用处。

如何使用呢？

Opcontrol –start –no-vmlinux

benchmarking

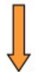
opcontrol –dump

opcontrol –shutdown

opreport -l /usr/local/bin/mysqld

执行完如下结果

```
# opreport -l /usr/local/bin/mysqld
samples   %        symbol name
83003     8.8858   String::copy(char const*, unsigned int, charset_info_st*,
charset_info_st*, unsigned int*)
79125     8.4706   MYSQLparse(void*)
68253     7.3067   my_wc_mb_latin1
55410     5.9318   my_pthread_fastmutex_lock
34677     3.7123   my_utf8_uni
18359     1.9654   MYSQLlex(void*, void*)
12044     1.2894   _ZL15get_hash_symbolPKcjb
11425     1.2231
_ZL20make_join_statisticsP4JOINP10TABLE_LISTP4ItemP16st_dynamic_array
```



- You can see quite a lot of CPU resources were spent for character conversions (latin1 <-> utf8)
- Disabling character code conversions on application side will improve performance (20% in this case)

```
samples   %        symbol name
83107    10.6202   MYSQLparse(void*)
68680     8.7765   my_pthread_fastmutex_lock
20469     2.6157   MYSQLlex(void*, void*)
13083     1.6719   _ZL15get_hash_symbolPKcjb
12148     1.5524   JOIN::optimize()
11529     1.4733
_ZL20make_join_statisticsP4JOINP10TABLE_LISTP4ItemP16st_dynamic_array
```