# 磁盘检查

## df /du

磁盘空间检查：df -hT / du 0sh

## iostate

磁盘使用率：iostat -x /dev/sda

## mount

磁盘挂载属性：mount -l

## lsblk

磁盘设备显示：lsblk

# 磁盘空间满导致mysql宕机

## 问题描述

MySQL宕机，错误日志显示如下：

[ERROR]InnoDB: Error number 28 means ‘No space left on device’

Congdanshi cong df -h看磁盘空间，磁盘空间是有剩余的，另外磁盘空间不足，重启以后也应该会出现问题，该问题仅出现了一次。

## 排查思路

1. 确认磁盘剩余空间
2. 确认磁盘配额
3. 检测磁盘坏道
4. 查看文件系统特性

## 排查过程

1. 从错误日志来看，是磁盘空间不足产生的redo日志写入失败。但一般除了磁盘满，redo是不会写失败的，因为redo是按照block来写的，每次写512字节，和扇区大小保持一致，所以redo不需要double write这种机制来保证写入的完整性。而且日志里面显示的字节大小是1536，是512的整数倍，只有可能是写磁盘失败了；
2. 查看磁盘空间和inode，发现空间有剩余，而inode显示为0，检查文件系统，发现使用的是btrfs系统，该系统是没有inode限制的，所以排除inode的原因；

df -i / df -hT

1. 检查磁盘坏块和坏道，发现没有坏块和坏道，检查命令badblocks -s -v /dev/sda1，使用btrfs自带的检查手段（check）检查磁盘，也没有问题；
2. 研究btrfs特性，btrfs需要利用cow技术来保证数据一致性，这项技术需要一个比较大的保留空间，看下面两个命令：

sudo btrfs fi show

sudo btrfs fi df /

Data,Single:total=133.01GB,used=121.05GB

磁盘的空间有397.99G，实际上的data空间只有133G，同时btrfs文件系统感知到空间满以后会自动执行类似btrfs balance start -v -dusage=0 /home（btrfs3.7.18后的特性）的命令来回收空间

## 结论

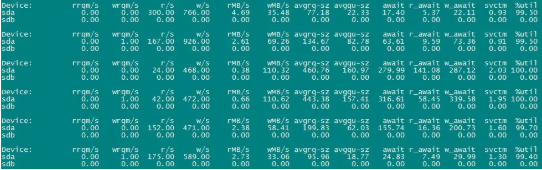
redo日志写入失败确实是由于空间不足导致的，而df -h看到的磁盘空间和btrfs真正使用的空间不一致，建议生产环境使用xfs，ext4文件系统。

# 磁盘IO高

## 问题描述

线上数据库突发严重告警，业务方反馈写入数据一直堵住，很多锁超时回滚了。

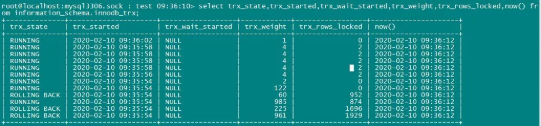
1. **系统负载，主要是磁盘I/O的负载数据**



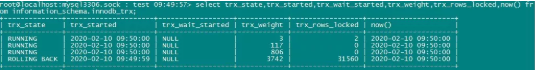
该服务器的磁盘是由6块2T SSD硬盘组成的RAID-5阵列。从上面的截图来看，I/O %util已经基本跑满了，io wait也非常高，很明显磁盘I/O压力太大了。那就再查查什么原因导致的这么高压力。

1. **活跃事务列表**

select trx\_state,trx\_started,trx\_wait\_started,trx\_\_weight,trx\_rows\_locked,now() from information\_schema.innodb\_trx;



可以看到，有几个活跃的事务代价很高，锁定了很多行。其中有两个因为太久超时被回滚了。



再看一次活跃事务列表，发现有个事务锁定的行更多了，说明活跃业务SQL的效率不太好，需要进行优化。这个算是原因之一，先记下。

**3、查看InnoDB状态**

执行SHOW ENGINE INNODB STATUS\G查看InnoDB状态，这里只展示了几个比较关键的地方：

...

0x7f8f700e9700 INNODB MONITOR OUTPUT

...

LATEST DETECTED DEADLOCK

------------------------

...

\*\*\* (2) TRANSACTION:

TRANSACTION 52970892097, ACTIVE 1 sec starting index read

mysql tables in use 2, locked 2

80249 lock struct(s), heap size 9691344, 351414 row lock(s),

undo log entries 30005

### 这里很明显，发生死锁的事务之一持有很多行锁，需要优化SQL

...

update a inner join b on a.uid=b.uid set

a.kid=if(b.okid=0,b.kid,b.okid),a.aid=b.aid where

a.date='2020-02-10'

...

TRANSACTIONS

------------

Trx id counter 52971738624

Purge done for trx's n:o < 52971738461 undo n:o < 0

state: running but idle

History list length 81

...

---TRANSACTION 52971738602, ACTIVE 0 sec inserting

mysql tables in use 1, locked 1

1 lock struct(s), heap size 1136, 0 row lock(s),

undo log entries 348

### 同样，也是有很多行锁

...

LOG

---

Log sequence number 565123876918590

Log flushed up to 565123858946703

Pages flushed up to 565121518602442

Last checkpoint at 565121518602442

...

**## 注意到Last checkpoint和LSN之间的差距非常大，约为2249MB**

**## 说明redo log的checkpoint延迟比较厉害，有可能是因为磁盘I/O太慢，**

**## 也有可能是因为产生的脏页太多太快，来不及刷新**

----------------------

BUFFER POOL AND MEMORY

----------------------

Total large memory allocated 201200762880

Dictionary memory allocated 130361859

Internal hash tables (constant factor + variable factor)

Adaptive hash index 3930999872 (3059599552 + 871400320)

Page hash 23903912 (buffer pool 0 only)

Dictionary cache 895261747 (764899888 + 130361859)

File system 16261960 (812272 + 15449688)

Lock system 478143288 (478120568 + 22720)

Recovery system 0 (0 + 0)

Buffer pool size 11795040

Buffer pool size, bytes 193249935360

Free buffers 7035886

Database pages 4705977

Old database pages 1737005

Modified db pages 238613

### 脏页比例约为2%，看着还好，而且还有挺多free page的

...

**4、查看MySQL的线程状态**

Show rpcesslists;

+---------+------+--------------+---------------------

|Command |Time |State |Info

+---------+------+--------------+---------------------

| Query | 1 | update | insert xxx

| Query | 0 | updating | update xxx

| Query | 0 | updating | update xxx

| Query | 0 | updating | update xxx

| Query | 0 | updating | update xxx

+---------+------+--------------+---------------------

可以看到几个事务都处于updating状态。意思是正在扫描数据并准备更新，肉眼可见这些事务状态时，一般是因为系统负载比较高，所以事务执行起来慢；或者该事务正等待行锁释放。

## 问题分析

分析上面的各种现场信息，我们可以得到以下几点结论：

1. 磁盘I/O压力很大。

先把阵列卡的cache策略改成WB，不过由于已经是SSD盘，这个作用并不大，只能申请更换成RAID-10阵列的新机器了，还需等待资源调配。

1. 需要优化活跃SQL，降低加锁代价

[root@test]> desc select \* from a inner join b on

a.uid=b.uid where a.date='2020-02-10';

+-------+--------+------+---------+----------+-------+----------+-----------------------+

| table | type | key | key\_len | ref | rows | filtered | Extra |

+-------+--------+------+---------+----------+-------+----------+-----------------------+

| a | ref | date | 3 | const | 95890 | 100.00 | NULL |

| b | eq\_ref | uid | 4 | db.a.uid | 1 | 100.00 | Using index condition |

+-------+--------+------+---------+----------+-------+----------+-----------------------+

[root@test]> select count(\*) from a inner join b on

a.uid=b.uid where a.date='2020-02-10';

+----------+

| count(\*) |

+----------+

| 40435 |

+----------+

1 row in set (0.22 sec)

执行计划看起来虽然能用到索引，但效率还是不高。检查了下，发现a表的uid列竟然没加索引。

3、InnoDB的redo log checkpoint延迟比较大，有2249MB之巨。先检查redo log的设置：

innodb\_log\_file\_size = 2G

innodb\_log\_files\_in\_group = 2

这个问题就大了，redo log明显太小，等待被checkpoint的redo都超过2G了，那肯定要疯狂刷脏页，所以磁盘I/O的写入才那么高，I/O %util和iowait也很高。

建议把redo log size调整成4G、3组。

innodb\_log\_file\_size = 4G

innodb\_log\_files\_in\_group = 2

此外，也顺便检查了InnoDB其他几个重要选项

innodb\_thread\_concurrency = 0

# 建议维持设置0不变

innodb\_max\_dirty\_pages\_pct = 50

# 由于这个实例每秒写入量较大，建议先调整到75，降低刷脏页的频率，

# 顺便缓解redo log checkpoint的压力。

# 在本案例，最后我们把这个值调整到了90。

**特别提醒**

从MySQL 5.6版本起，修改redo log设置后，实例重启时会自动完成redo log的再次初始化，不过前提是要先干净关闭实例。因此建议在第一次关闭时，修改以下两个选项：

innodb\_max\_dirty\_pages\_pct = 0

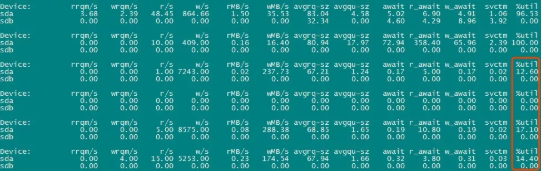
innodb\_fast\_shutdown = 0

并且，再加上一个新选项，防止实例启动后，会有外部应用连接进来继续写数据：

skip-networking

在确保所有脏页（上面看到的Modified db pages为0）都刷盘完毕后，并且redo log也都checkpoint完毕（上面看到的Log sequence number和Last checkpoint at\*\*值相等），此时才能放心的修改 innodb\_log\_file\_size 选项配置并重启。确认生效后再关闭 skip-networking 选项对业务提供服务。

经过一番优化调整后，再来看下服务器和数据库的负载。



可以看到，服务器的磁盘I/O压力再也不会那么大了，数据库中也不会频繁出现大量行锁等待或回滚的事务了。

[root@yejr.me]> SHOW ENGINE INNODB STATUS\g

Log sequence number 565749866400449

Log flushed up to 565749866400449

Pages flushed up to 565749866318224

Last checkpoint at 565749866315740

[root@yejr.me]> SHOW ENGINE INNODB STATUS\g

Log sequence number 565749866414660

Log flushed up to 565749866400449

Pages flushed up to 565749866320827

Last checkpoint at 565749866315740

[root@yejr.me]> SHOW ENGINE INNODB STATUS\g

Log sequence number 565749866414660

Log flushed up to 565749866414660

Pages flushed up to 565749866322135

Last checkpoint at 565749866315740

[root@yejr.me]> select (565749866400449-565749866315740)/1024;

+----------------------------------------+

| (565749866400449-565749866315740)/1024 |

+----------------------------------------+

| 82.7236 |

+----------------------------------------+

1 row in set (0.00 sec)

[root@yejr.me]> select (565749866414660-565749866315740)/1024;

+----------------------------------------+

| (565749866414660-565749866315740)/1024 |

+----------------------------------------+

| 96.6016 |

+----------------------------------------+

很明显，redo log checkpoint lag几乎没有了。

# 线上IO问题

## 问题描述

某业务CDB实例，每天在特地时间段内（00:07:00 - 00:08:00左右）机器对应IO监控出现写入尖刺，且主从实例都有类似现象，从机器监控可以看到，问题确实存在。

不仅master，进行同步的slave上有相同的现象，业务方希望找到导致该IO尖刺问题稳定出现的原因。

## 问题分析

首先确定问题来源，上图所示监控为机器级别，机器IO写入负载是否来源于mysqld进程？如果来源于mysqld进程，是来自于mysqld进程的哪一部分写入操作引起？

为了获取IO来源，在slave机上部署mysqld实例监控，以及iotop采集监控，获取对应时间段更详细的相关信息，抓取对应时间段进行IO写入的进程（线程），同时观察对应时间段mysql实例状态。

这段时间内的较大IO写入线程号为：（截取部分记录）

时间 线程号 进程名 读取速度 写入速度

00:07:34 145378 be/4 mysql201 139.10 K/s 263111.57 K/s

00:07:35 145378 be/4 mysql201 124.11 K/s 249703.84 K/s

00:07:36 145378 be/4 mysql201 120.23 K/s 289920.70 K/s

00:07:39 145378 be/4 mysql201 5168.09 K/s 875194.69 K/s

通过slave上iotop采集到的统计信息，可以看出较大写入来源为145378这一线程，确实来自于mysql进程，该时间段内没有抓到其他大量写入的记录，同时该实例slave mysql为单机独占，可以基本确定写入来源为mysql中145378这个线程，那么这个线程是哪一个线程呢？

145378这个线程号正是mysqld slave的线程，而且为单线程回放的sql线程。

Thread 85 (Thread 0x7f68c4c4c700 (LWP 145378)):

#0 0x00007fa2badd3945 in pthread\_cond\_wait@@GLIBC\_2.3.2 () from /lib64/libpthread.so.0

#4 exec\_relay\_log\_event (rli=0x1771c43c8, thd=0x7f68b0000990)

#5 handle\_slave\_sql (arg=0x1771c3010)

通过进一步搜索监控记录，可以发现，其实sql线程引入大量写入IO不仅存在于这个时间段，在其他时间段也有较高写入的记录（超过100MB/s），在00:08:00左右持续时间相对较长。这个现象反应出该问题可能贯穿于整个执行过程，只是其他时间段没有这么明显。

通过mysql实例的监控可，可以看到mysql主要进行的操作为insert，slave mysql主要在进行单线程回放，执行这些insert操作，操作数量会有一些增长，每秒操作数不稳定，峰值可能达到4000左右。

从对于slave实例的新增监控可以得到结论，这段时间的主要写入来源确实为mysqld进程，且为mysqld的sql线程，那么问题转换为：

在该业务模型下，进行单线程回放的mysql slave sql thread， 为什么会在一些时段产生较大的写入IO？产生这个现象的时候，mysql在做什么事情？

为了更好的分析问题，同时不影响线上业务，在另外一台实验机器上单独搭建mysql 5.7的slave，连到源实例的master作为slave，问题现象也可以复现，为了获取更多信息，打开mysql 5.7的performance\_schema，在实验机器的slave机监控上，依然能看到该问题存在。

现在我们需要分析一下，SQL线程回放，可能产生哪些IO写（注意其他线程的IO不会记录到SQL线程头上，例如page cleaner flush），一条SQL语句回放过程中，可能经历的路径上有哪些操作会引发IO操作。

一个slave mysql读取relay log进行日志回放，首先想到回放执行语句，可能由此引发下列写入IO：

mysql server binlog日志记录，即回放过程中语句写入的本地binlog。

innodb redo undo日志记录。

用户态page flush（free page吃紧，需要自行flush获取可用free page）。

relay log purge，删除文件。默认每当slave回放完一个完整relay log，会将对应文件进行删除。

那么，为了找到真正引发写入的来源，需要对于上述来源逐步进行分析、排除，对可能原因进行验证。

首先在5.7 slave上配置log\_slave\_updates=OFF，使得回放过程不记录binlog日志，问题现象依旧，排除binlog写入。

分钟级监控显示，master实例在对应时间段insert数量有一定幅度增加：

为了保证free page够用，调大了buffer pool，确保free page足够使用，另外关闭relay log purge功能，使得SQL线程不会触发删除relay log文件，问题现象依旧，排除清理relay以及用户态page flush。

通过sys schema统计值发现，对于文件写入，count write统计值较大的记录主要来源于redo log（例如ib\_logfile0写入7439952次，且总量为57.47GB），但是innodb自身对于log写入的单位时间统计值显示却不大。

/data1/mysql\_root/log/20120/ib\_logfile0 5 4.00 KiB 819 bytes 7439952 57.47 GiB 8.10 KiB 57.47 GiB 100.00

/data1/mysql\_root/log/20120/ib\_logfile1 2 64.50 KiB 32.25 KiB 3025254 23.39 GiB 8.11 KiB 23.39 GiB 100.00

为了进一步排除干扰，修改mysql实现关闭redo log写入，替换mysql 5.7实验版本，统计值显示确实redo文件几乎没有写入增长，问题依旧，排除redo写入造成，再看ibdata相关记录增长，也非常有限，也可以排除undo文件写入。

到目前为止，通常能想到的用户态写入（例如sql线程回放执行一个事务），都可以排除掉，还有什么原因可以sql线程造成大量写入呢？需要重新整理一下思路。

再回到业务本身，看一下业务的库表结构模型和数据规模，表结构大体如下：

FRFrom int(10) unsigned NOT NULL DEFAULT ‘0’,

FRtBody mediumblob NOT NULL,

FPSQL varchar(20480) NOT NULL,

FBody mediumblob NOT NULL,

FResTime bigint(20) NOT NULL DEFAULT ‘0’,

FCime bigint(20) NOT NULL DEFAULT ‘0’,

FInt1 int(10) unsigned NOT NULL DEFAULT ‘0’,

FInt2 int(10) unsigned NOT NULL DEFAULT ‘0’,

FInt3 int(10) unsigned NOT NULL DEFAULT ‘0’,

FInt4 int(10) unsigned NOT NULL DEFAULT ‘0’,

FInt5 int(10) unsigned NOT NULL DEFAULT ‘0’,

FChar1 varchar(256) NOT NULL DEFAULT ‘’,

FChar2 varchar(256) NOT NULL DEFAULT ‘’,

FChar3 varchar(512) NOT NULL DEFAULT ‘’,

FChar4 varchar(1024) NOT NULL DEFAULT ‘’,

FChar5 varchar(1024) NOT NULL DEFAULT ‘’,

FExt blob,

FGrp2EvtNo bigint(20) DEFAULT NULL,

PRIMARY KEY (FEventNo)

表数量很多，超过1W张表，且单表数据量不大

表结构没有其他索引，只有主键

包含blob字段

从relay log分析插入语句，blob每次都有较大的数据量进行插入

表数量较多，插入操作，blob字段，并发插入且插入操作分散到各个表，这几点看起来有些关联。在关掉了包括binlog，relay purge，redo log等多个写入之后，再抓一下sql线程回放的堆栈，看一下写入调用的来源：

#0 os\_aio\_func (type=..., mode=mode@entry=24,

name=0x7ee4493f97b8 "./DB\_xxx/xxx\_36.ibd", file=..., buf=buf@entry=0x7ee453ed4000, offset=offset@entry=122683392, n=n@entry=1048576, read\_only=read\_only@entry=false, m1=m1@entry=0x0, m2=m2@entry=0x0)

#1 0x0000000001187922 in fil\_write\_zeros (node=0x7ee4493f5338, node=0x7ee4493f5338, read\_only\_mode=<optimized out>,

len=4194304, start=121634816, page\_size=<optimized out>)

#2 fil\_space\_extend (space=space@entry=0x7ee448d8dbd8, size=<optimized out>)

#3 0x00000000007701d1 in fsp\_try\_extend\_data\_file (space=space@entry=0x7ee448d8dbd8, header=header@entry=0x7ef4a4ff0026 "",

#4 0x000000000118f0df in fsp\_reserve\_free\_extents (n\_reserved=n\_reserved@entry=0x7f20868f4fa0, space\_id=9054, n\_ext=3,

alloc\_type=alloc\_type@entry=FSP\_NORMAL, mtr=mtr@entry=0x7f20868f5890, n\_pages=n\_pages@entry=2)

#5 0x00000000010e6394 in btr\_cur\_pessimistic\_insert (flags=flags@entry=0, cursor=cursor@entry=0x7f20868f5150,

offsets=offsets@entry=0x7f20868f50b0, heap=heap@entry=0x7f20868f50a0, entry=entry@entry=0x7ee453143488,

rec=rec@entry=0x7f20868f5570, big\_rec=big\_rec@entry=0x7f20868f5090, n\_ext=n\_ext@entry=0, thr=thr@entry=0x7ee4544623f0,

mtr=mtr@entry=0x7f20868f5890)

回顾一下基于主键索引的插入操作，对于b+树，如果插入的record的较大（例如很大的blob），可能会触发分裂操作。类似

对于innodb而言，插入的过程中，首先尝试乐观插入索引，如果空间大小不够，再尝试悲观插入，悲观插入首先保证表空间大小足够（ibd文件 innodb\_file\_per\_table=ON，每个表对应一个文件），这里用户场景，正是每个表一个ibd文件，那么16000张表总共有16000个ibd文件。如果空间不够，尝试对于ibd文件进行扩展，扩展逻辑如下：

每次扩展4个extent（每个extent包含 16kb\*64 = 1MB大小供64个data page），即每次扩展形成4MB，业务场景包含16000多个表，且每次插入数据量相对固定，表结构相同，插入目标表随机分散，所以很多表大小，和后续操作，非常均衡可以看作是齐头并进。

从slave实例监控来看，insert在对应时间段超过1000个每秒，最多可以达到4000个每秒。假设相同数据量模型，其中500个表同时扩展大小，这一秒内，可能同时产生500\*4MB = 2GB左右的写入。为了印证这个写入来源，继续在innodb层添加日志跟踪，对于idb文件扩展加入以下逻辑进行日志跟踪：

每100MB扩展数据量，查看累计时间，如果累计时间在1秒以内，打印一条warning日志，且sleep 0.5s。

查看对应时间段新增日志（+8为北京时间），

对应时间扩展很频繁，除去sleep时间，大约0.1-0.2秒能够产生100MB的扩展写入，IO尖刺数据量基本吻合，与iotop抓取到的大io写入也基本吻合。至此我们基本可以得出问题结论。

## 问题结论

业务模型比较特殊，多个包含BLOB字段的小表（超过16000），业务模型为insert，写入目标表分散，在某些时间段会并发insert（平均2000个左右每秒，峰值4000个每秒），由于BLOB字段占用空间较大，各个表使用分散，导致频繁同时触发分裂，进而导致底层data page扩展，使得底层多个ibd文件同时并发extend，初始化data page产生某些时间段较大IO，反应到机器监控上为某些时间段IO尖刺

## 解决方法

业务层面，对于类似场景，考虑合并表数量，减少并发扩展带来的写入压力，可以一定程度缓解IO尖刺。

MySQL层面，考虑更加智能的数据文件扩展算法，适配上述场景。减少扩展的并发性。

MySQL层面，考虑用户指定初始化表空间大小，提前预分配和初始化，避免动态扩展。

# 负载过高问题总结

## OS层面检查确认

登入服务器后，我们的目的是首先要确认当前到底是哪些进程引起的负载高，以及这些进程卡在什么地方，瓶颈是什么。

通常来说，服务器上最容易成为瓶颈的是磁盘I/O子系统，因为它的读写速度通常是最慢的。即便是现在的PCIE SSD，其随机I/O读写速度也是不如内存来得快。当然了，引起磁盘I/O慢得原因也有多种，需要确认哪种引起的。

### 整体负载

第一步，我们一般先看整体负载如何，负载高的话，肯定所有的进程跑起来都慢。

**可以执行指令w或者sar -q 1来查看负载数据**，例如：

[\*\*\*\*@\*\*\*\* ]# w

11:52:58 up 702 days, 56 min, 1 user, load average: 7.20, 6.70, 6.47

USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT

root pts/0 1.xx.xx.xx 11:51 0.00s 0.03s 0.00s w

或者sar -q的观察结果：

[\*\*\*\*@\*\*\*\*]# sar -q 1

Linux 2.6.32-431.el6.x86\_64 (test) 01/13/2016 \_x86\_64\_ (24 CPU)

02:51:18 PM runq-sz plist-sz ldavg-1 ldavg-5 ldavg-15 blocked

02:51:19 PM 4 2305 6.41 6.98 7.12 3

02:51:20 PM 2 2301 6.41 6.98 7.12 4

02:51:21 PM 0 2300 6.41 6.98 7.12 5

02:51:22 PM 6 2301 6.41 6.98 7.12 8

02:51:23 PM 2 2290 6.41 6.98 7.12 8

load average大意表示当前CPU中有多少任务在排队等待，等待越多说明负载越高，跑数据库的服务器上，一般load值超过5的话，已经算是比较高的了。

引起load高的原因也可能有多种：

1、某些进程/服务消耗更多CPU资源（服务响应更多请求或存在某些应用瓶颈）；

2、发生比较严重的swap（可用物理内存不足）；

3、发生比较严重的中断（因为SSD或网络的原因发生中断）；

4、磁盘I/O比较慢（会导致CPU一直等待磁盘I/O请求）；

### 子系统瓶颈

这时我们可以执行下面的命令来判断到底瓶颈在哪个子系统：

[\*\*\*\*@\*\*\*\*]# top

top - 11:53:04 up 702 days, 56 min, 1 user, load average: 7.18, 6.70, 6.47

Tasks: 576 total, 1 running, 575 sleeping, 0 stopped, 0 zombie

Cpu(s): 7.7%us, 3.4%sy, 0.0%ni, 77.6%id, 11.0%wa, 0.0%hi, 0.3%si, 0.0%st

Mem: 49374024k total, 32018844k used, 17355180k free, 115416k buffers

Swap: 16777208k total, 117612k used, 16659596k free, 5689020k cached

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND

14165 mysql 20 0 8822m 3.1g 4672 S 162.3 6.6 89839:59 mysqld

40610 mysql 20 0 25.6g 14g 8336 S 121.7 31.5 282809:08 mysqld

49023 mysql 20 0 16.9g 5.1g 4772 S 4.6 10.8 34940:09 mysqld

很明显是前面两个mysqld进程导致整体负载较高。

而且，从Cpu(s)这行的统计结果也能看的出来，%us和%wa的值较高，表示当前比较大的瓶颈可能是在用户进程消耗的CPU以及磁盘I/O等待上。

### 磁盘I/O

我们先分析下磁盘I/O的情况。

执行sar -d确认磁盘I/O是否真的较大：

[\*\*\*\*@\*\*\*\*]# sar -d 1

Linux 2.6.32-431.el6.x86\_64 (test) 01/13/2016 \_x86\_64\_ (24 CPU)

11:54:32 AM dev8-0 5338.00 162784.00 1394.00 30.76 5.24 0.98 0.19 100.00

11:54:33 AM dev8-0 5134.00 148032.00 32365.00 35.14 6.93 1.34 0.19 100.10

11:54:34 AM dev8-0 5233.00 161376.00 996.00 31.03 9.77 1.88 0.19 100.00

11:54:35 AM dev8-0 4566.00 139232.00 1166.00 30.75 5.37 1.18 0.22 100.00

11:54:36 AM dev8-0 4665.00 145920.00 630.00 31.41 5.94 1.27 0.21 100.00

11:54:37 AM dev8-0 4994.00 156544.00 546.00 31.46 7.07 1.42 0.20 100.00

再利用iotop确认到底哪些进程消耗的磁盘I/O资源最多：

[\*\*\*\*@\*\*\*\*]# iotop

Total DISK READ: 60.38 M/s | Total DISK WRITE: 640.34 K/s

TID PRIO USER DISK READ DISK WRITE SWAPIN IO> COMMAND

16397 be/4 mysql 8.92M/s 0.00B/s 0.00% 94.77% mysqld --basedir=/usr/local/m~og\_3320/mysql.sock --port=3320

7295 be/4 mysql 10.98M/s 0.00B/s 0.00% 93.59% mysqld --basedir=/usr/local/m~og\_3320/mysql.sock --port=3320

14295 be/4 mysql 10.50M/s 0.00B/s 0.00% 93.57% mysqld --basedir=/usr/local/m~og\_3320/mysql.sock --port=3320

14288 be/4 mysql 14.30M/s 0.00B/s 0.00% 91.86% mysqld --basedir=/usr/local/m~og\_3320/mysql.sock --port=3320

14292 be/4 mysql 14.37M/s 0.00B/s 0.00% 91.23% mysqld --basedir=/usr/local/m~og\_3320/mysql.sock --port=3320

可以看到，端口号是3320的实例消耗的磁盘I/O资源比较多，那就看看这个实例里都有什么查询在跑。

## MySQL层面检查确认

首先看下当前都有哪些查询在运行：

[\*\*\*\*@\*\*\*\*]> mysqladmin pr|grep -v Sleep

+----+----+----------+----+-------+-----+--------------+-----------------------------------------------------------------------------------------------+

| Id |User| Host | db |Command|Time | State | Info |

+----+----+----------+----+-------+-----+--------------+-----------------------------------------------------------------------------------------------+

| 25 | x | 10.x:8519 | db | Query | 68 | Sending data | select max(Fvideoid) from (select Fvideoid from t where Fvideoid>404612 order by Fvideoid) t1 |

| 26 | x | 10.x:8520 | db | Query | 65 | Sending data | select max(Fvideoid) from (select Fvideoid from t where Fvideoid>484915 order by Fvideoid) t1 |

| 28 | x | 10.x:8522 | db | Query | 130 | Sending data | select max(Fvideoid) from (select Fvideoid from t where Fvideoid>404641 order by Fvideoid) t1 |

| 27 | x | 10.x:8521 | db | Query | 167 | Sending data | select max(Fvideoid) from (select Fvideoid from t where Fvideoid>324157 order by Fvideoid) t1 |

| 36 | x | 10.x:8727 | db | Query | 174 | Sending data | select max(Fvideoid) from (select Fvideoid from t where Fvideoid>324346 order by Fvideoid) t1 |

+----+----+----------+----+-------+-----+--------------+-----------------------------------------------------------------------------------------------+

可以看到有不少慢查询还未完成，从slow query log中也能发现，这类SQL发生的频率很高。

这是一个非常低效的SQL写法，导致需要对整个主键进行扫描，但实际上只需要取得一个最大值而已，从slow query log中可看到：

Rows\_sent: 1 Rows\_examined: 5502460

每次都要扫描500多万行数据，却只为读取一个最大值，效率非常低。

经过分析，这个SQL稍做简单改造即可在个位数毫秒级内完成，原先则是需要150-180秒才能完成，提升了N次方。

改造的方法是：对查询结果做一次倒序排序，取得第一条记录即可。而原先的做法是对结果正序排序，取最后一条记录，汗啊。。。

## 总结

在这个例子中，产生瓶颈的原因比较好定位，SQL优化也不难，实际线上环境中，通常有以下几种常见的原因导致负载较高：

1、一次请求读写的数据量太大，导致磁盘I/O读写值较大，例如一个SQL里要读取或更新几万行数据甚至更多，这种最好是想办法减少一次读写的数据量；

2、SQL查询中没有适当的索引可以用来完成条件过滤、排序（ORDER BY）、分组（GROUP BY）、数据聚合（MIN/MAX/COUNT/AVG等），可以添加索引或者进行SQL改写；

3、瞬间突发有大量请求，这种一般只要能扛过峰值就好，保险起见还是要适当提高服务器的配置，万一峰值抗不过去就可能发生雪崩效应；

4、因为某些定时任务引起的负载升高，比如做数据统计分析和备份，这种对CPU、内存、磁盘I/O消耗都很大，最好放在独立的slave服务器上执行；

5、服务器自身的节能策略发现负载较低时会让CPU降频，当发现负载升高时再自动升频，但通常不是那么及时，结果导致CPU性能不足，抗不过突发的请求；

6、使用raid卡的时候，通常配备BBU（cache模块的备用电池），早期一般采用锂电池技术，需要定期充放电（DELL服务器90天一次，IBM是30天），我们可以通过监控在下一次充放电的时间前在业务低谷时提前对其进行放电，不过新一代服务器大多采用电容式电池，也就不存在这个问题了。

7、文件系统采用ext4甚至ext3，而不是xfs，在高I/O压力时，很可能导致%util已经跑到100%了，但iops却无法再提升，换成xfs一般可获得大幅提升；

8、内核的io scheduler策略采用cfq而非deadline或noop，可以在线直接调整，也可获得大幅提升。