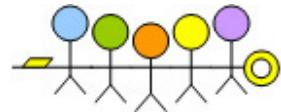


MongoDB 测试报告

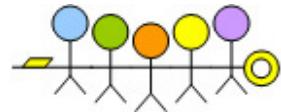


目录

MongoDB 测试报告	1
测试环境	8
加压机、	8
服务器	8
监测工具	8
Mongostat	8
Zabbix	9
软件版本	9
测试用数据结构	9
词库	9
结构	10
范例	11
测试	12
i1、不同索引对于插入性能的影响	12
测试描述：	12
测试数据	13
猜测	13
结论	13
i2、1亿级别数据无索引插入性能测试	14
测试描述：	14
测试数据	14
结论：	15

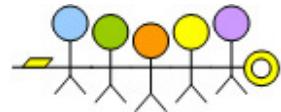


i3、5000万级别数据全索引插入性能测试.....	16
测试描述：	16
测试数据	16
结论：	17
i4、2亿数据双索引插入（1亿基础上）性能测试.....	18
测试描述：	18
测试数据	18
结论.....	19
i5、2亿数据双索引插入（3亿基础上）性能测试.....	19
测试描述：	19
测试数据	20
结论.....	20
i6、Shard 插入性能测试.....	20
测试描述：	20
测试数据	21
结论.....	21
i7、五组大数据量插入性能对比.....	22
测试描述：	22
测试数据	22
结论.....	22
i8、不同数据量占用磁盘空间大小对比.....	23
测试描述：	23
测试数据	23



结论.....	24
i9、GridFS 和二进制插入性能对比.....	24
测试描述：	24
测试数据	24
二进制插入的 java 代码-范例	25
结论.....	25
猜测.....	25
q1、针对不同数据类型查询的性能差别.....	26
测试描述：	26
测试数据	26
结论.....	26
q2、并发数不同对于查询性能的影响.....	27
测试描述：	27
测试数据	27
结论.....	28
猜测.....	28
q3、Limit 对于查询性能的影响.....	28
测试描述：	28
测试数据	28
结论.....	29
q4、skip 对于查询性能的影响.....	29
测试描述：	29
测试数据	29

结论.....	29
q5、1亿数据随机全查性能测试.....	30
测试描述：	30
测试数据.....	30
Zabbix 监测数据.....	30
结论.....	31
q6、1亿数据随机查前 1000 万性能测试.....	31
测试描述：	31
测试数据.....	32
Zabbix 监测数据.....	32
结论.....	33
q7、热数据量大小 (< 内存大小) 对于查询性能的影响	34
测试描述：	34
测试数据.....	34
结论.....	34
q8、热数据进入内存过程对于查询性能的影响	34
测试描述：	34
测试数据.....	35
结论.....	36
q9、1-3-5 亿数据量查询前 1000 万性能对比	36
测试描述：	36
测试数据	36
结论.....	37



q10、1-3-5亿数据量查询前1亿性能对比	37
测试描述：	37
测试数据	37
结论	38
q11、1-3-5亿数据不同热数据量查询性能对比	38
测试描述：	38
测试数据	38
结论	40
q12、5亿数据量查询不同数据量性能对比	41
测试描述：	41
测试数据	41
结论	42
q13、Shard(双机) 查询性能测试	42
测试描述：	42
测试数据	42
结论	43
q14、Shard(双机) 和单服务器的查询性能对比	44
测试描述：	44
测试数据	44
结论	44
q15、CPU与查询性能的关系	44
测试描述：	44
测试数据	45



结论.....	45
o1、不同情况建立索引的对比	46
现象描述：	46
结论.....	46
o2、repairDatabase 的现象	46
现象描述	46
结论.....	47
测试结论	47
插入.....	47
查询.....	48
没有解决的问题.....	49

测试环境

加压机、

系统：WindowsXP SP3，32位

内存：4G

CPU：双核/2.2GHz

IO：33M/秒

服务器

系统：centos 系统，64位

CPU：4 核/2.1GHz (除了进行 CPU 个数测试的时候)

IO 上限：未测得，未达到，未成为限制

各服务器的内存和磁盘大小因需要各不一样

监测工具

Mongostat

1) 监测速度

2) 监测 IO 数值

3) 监测连接数 con



4) 监测内存 res

5) 监测 locked

Zabbix

1) 监测内存

2) 监测 CPU 情况

3) 监测 IO 数值

4) SWAP

软件版本

服务器 MongoDB 版本 : 2.4.8

加压机 Java jdk 版本 : 1.7.0_45

MongoDB Java 驱动版本 : 2.9.3

测试用数据结构

词库

词库 1 : dict , 从英语字典里抓取 7998 个词汇

词库 2 : color , 自己建立 , 10 个词



结构

“_id”	系统生成 , 12 字节,	
数字】 “numb”	自增的	
字符串】 “name”	从词库 1 中随机	
长字符串】 “describe”	128 字节 , String , 固定的	
时间】 "time"	ISODate("2013-11-19T08:23:03.468Z")	
bool】 “flag”	随机,	
数值】 “price”	20-1020 之间随机的随机数	
嵌套对象】 “parameter”	数值】 “size”	0-50 之间随机数,
	字符串】 “logo”	词库 1 中随机前 100
	数组】 “color”	长度从 2-10 随机 , 内容从词库 2 随机
数组】 “category”	长度从 2-10 随机 , 内容从词库 1 随机	
图片】 “image”	1.04M , 名字和 numb 相同	



范例

```
"_id" : ObjectId("528d6cd5621e8e90e5ae7a7f"),  
  
"number" : 0,  
  
"name" : "mutton",  
  
"describe" : "The most distant way in the world is not the way from birth to the end.  
  
It is when I sit near you that you don't understand I love you.  
  
The most distant way in the world is not that you're not sure I love you.  
  
It is when my love is bewildering the soul, but I can't speak it out.",  
  
"time" : ISODate("2013-11-21T02:15:49.093Z"),  
  
"flag" : 0,  
  
"price" : 975,  
  
"parameter" : {  
  
    "size" : 1,  
  
    "logo" : "accelerate",  
  
    "color" : [  
  
        "black", "yellow", "black", "blue", "pink", "indigo", "red",
```



```
]  
},  
"category" : [  
    "fascination", "credit", "comparison", "island", "grandfather"  
]
```

测试

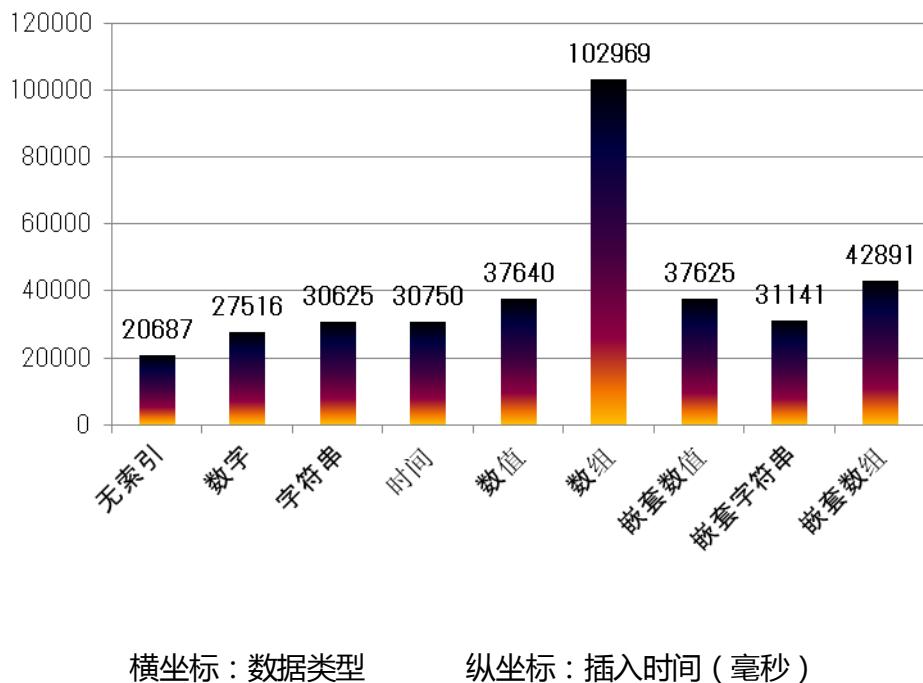
i1、不同索引对于插入性能的影响

测试描述：

4g 内存，单线程，单索引，从 0 插入 20 万条数据



测试数据



横坐标：数据类型

纵坐标：插入时间（毫秒）

猜测

- 1、数组列耗时较长的可能原因：数组内字段比较多，同时每个元素的变化范围也比较大。
- 2、可能每个索引的建立对于性能的影响主要从三个方面：一个是该列数据的分布范围（比如，从1万中可能中变化的要慢于从10个可能中变化的），另外就是该列数据包含的数据数量（数组就明显慢于字符串），另数字和时间比较快可能和这两列的值是比较有规律的有关。

结论

建立索引后，插入性能有明显下降，但是，不同数据类型的索引对于插入性能影响不大。

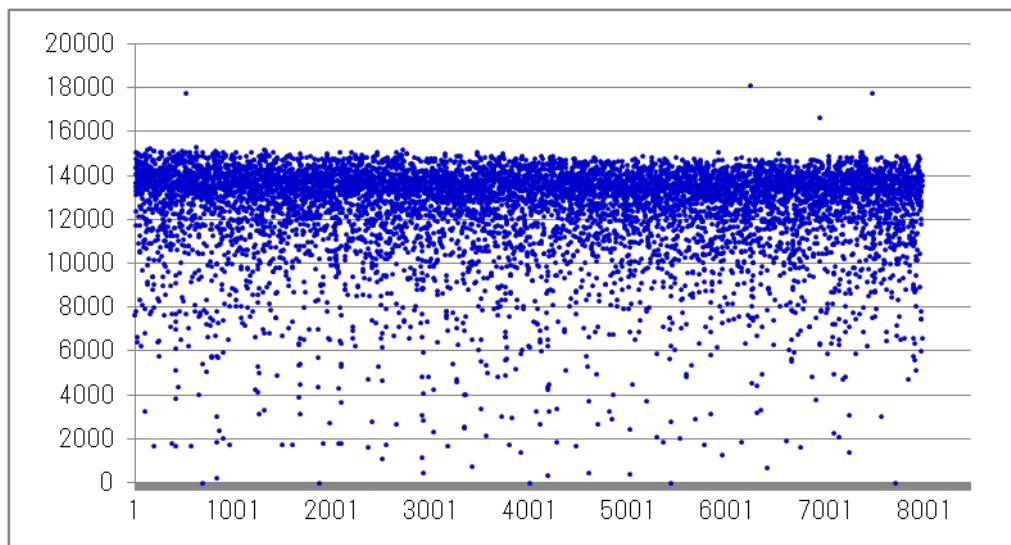


i2、1亿级别数据无索引插入性能测试

测试描述：

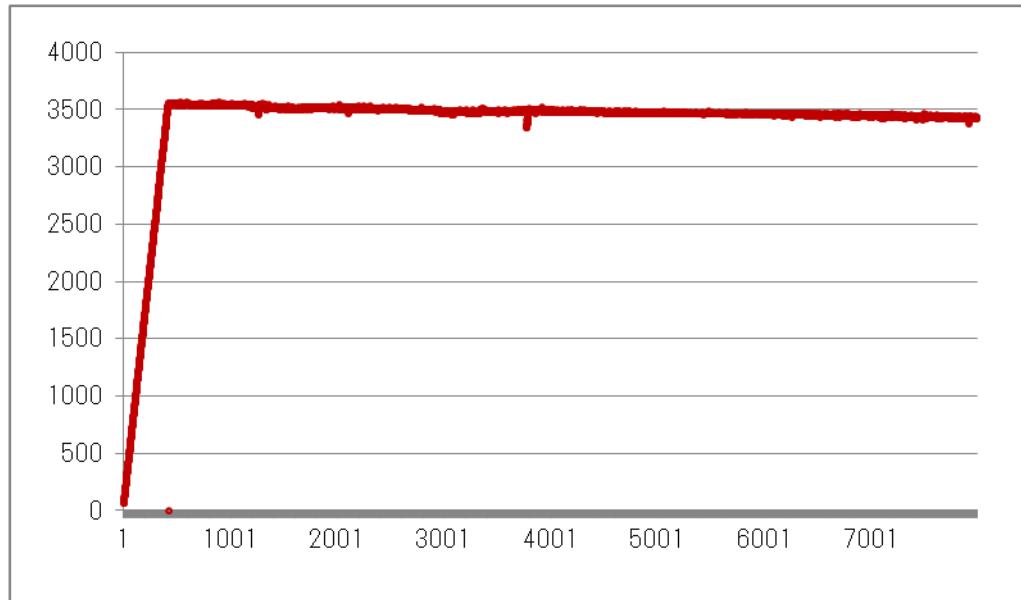
单线程插入 1 亿条数据，插入前数据库无索引、无数据。

测试数据



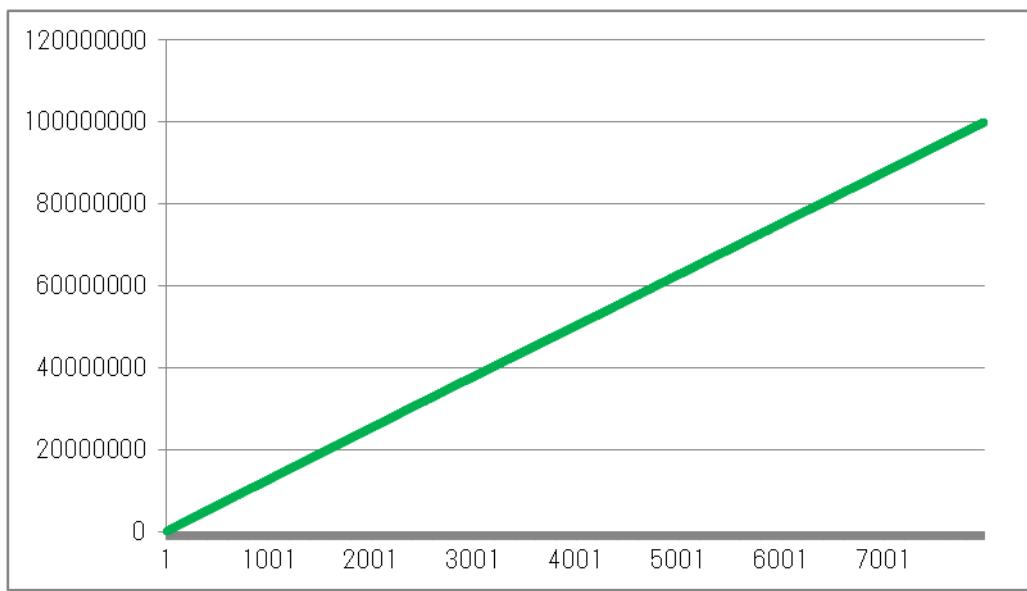
横坐标：时间(秒)

纵坐标：插入速度 (个/秒)



横坐标：时间(秒)

纵坐标：内存占用量(MB)(4G)

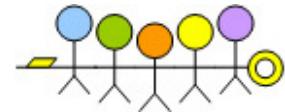


横坐标：时间(秒)

纵坐标：总数据量（个数）

结论：

无索引状态下，内存状态(装满与否)对于插入速度没有影响，插入速度宏观上稳定，有明显震荡。

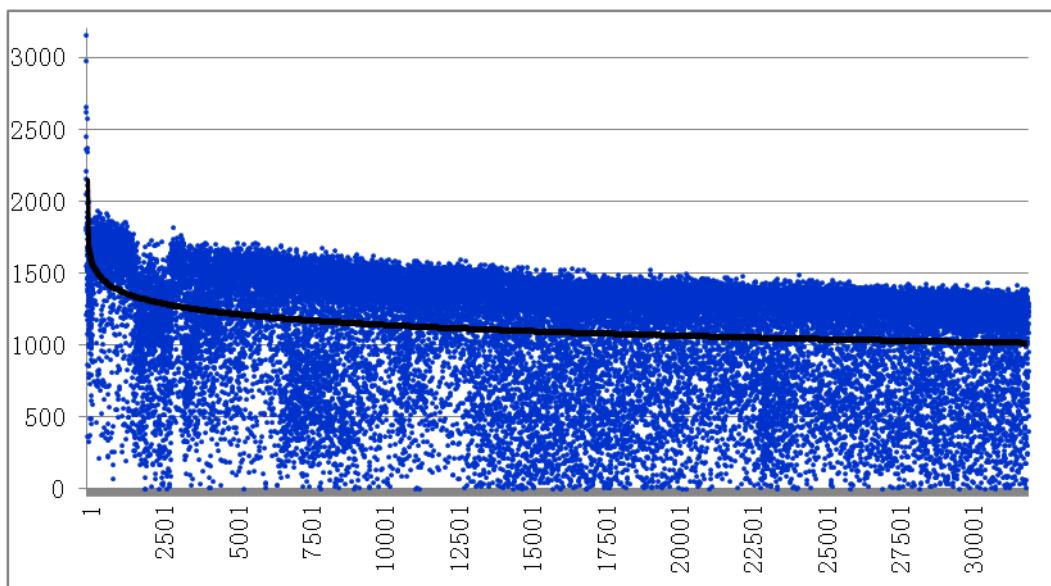


i3、5000 万级别数据全索引插入性能测试

测试描述：

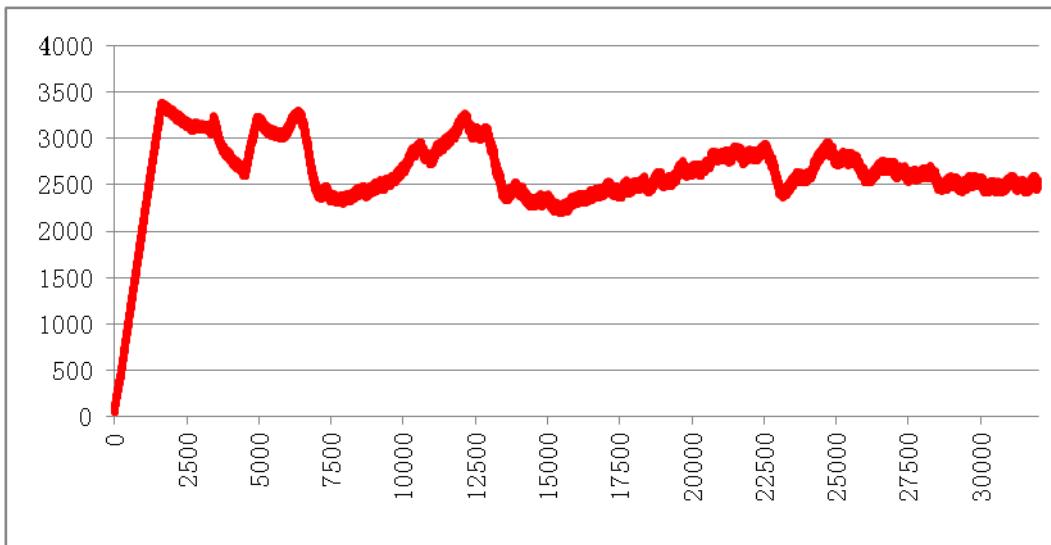
单线程插入 5000 万条数据，插入前建立全索引 (number, name, time, flag, price, parameter.size, parameter.logo, parameter.parameter.color, category)，无数据，原计划插入 1 亿，后因速度太慢，时间太长终止于五千万。

测试数据



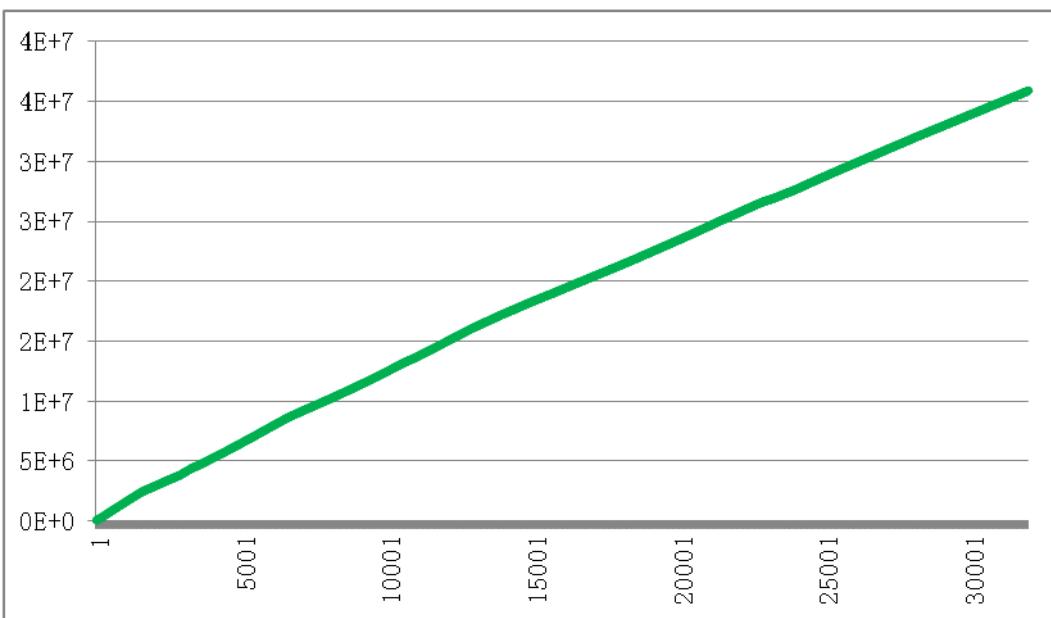
横坐标：时间(秒)

纵坐标：插入速度 (个/秒)



横坐标：时间(秒)

纵坐标：内存占用量(MB)(4G)

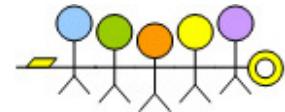


横坐标：时间(秒)

纵坐标：总数据量（个数）

结论：

有索引的状态下，内存状态(装满与否)对于插入速度有影响，但是影响不大；插入速度随着数据量的增加而缓慢下降，震荡一直存在。

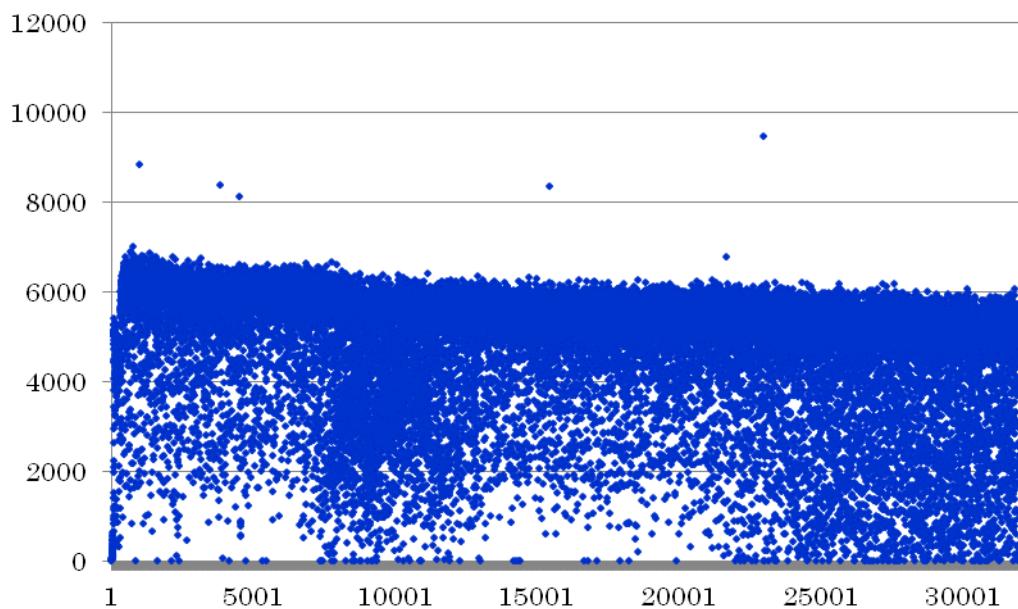


i4、2亿数据双索引插入（1亿基础上）性能测试

测试描述：

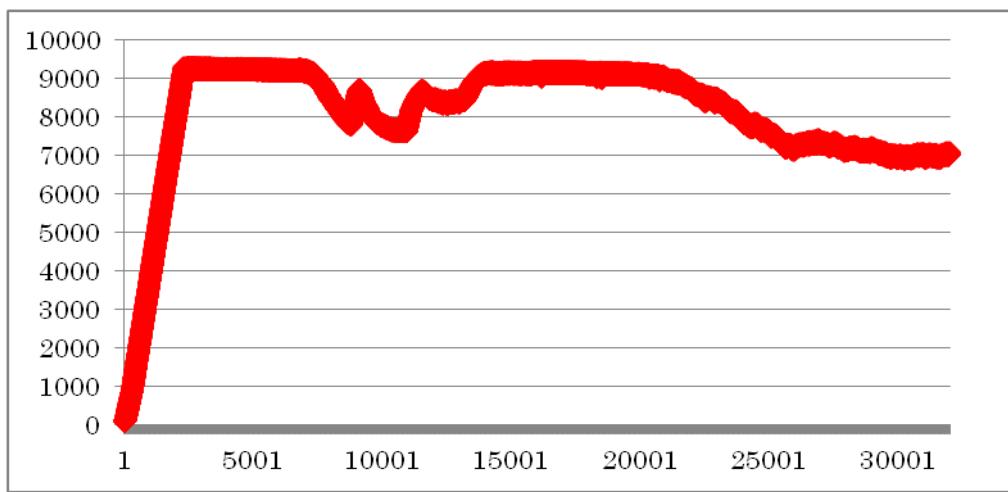
在1亿数据基础上单线程插入2亿条数据，有 双索引：number、name，内存10g

测试数据



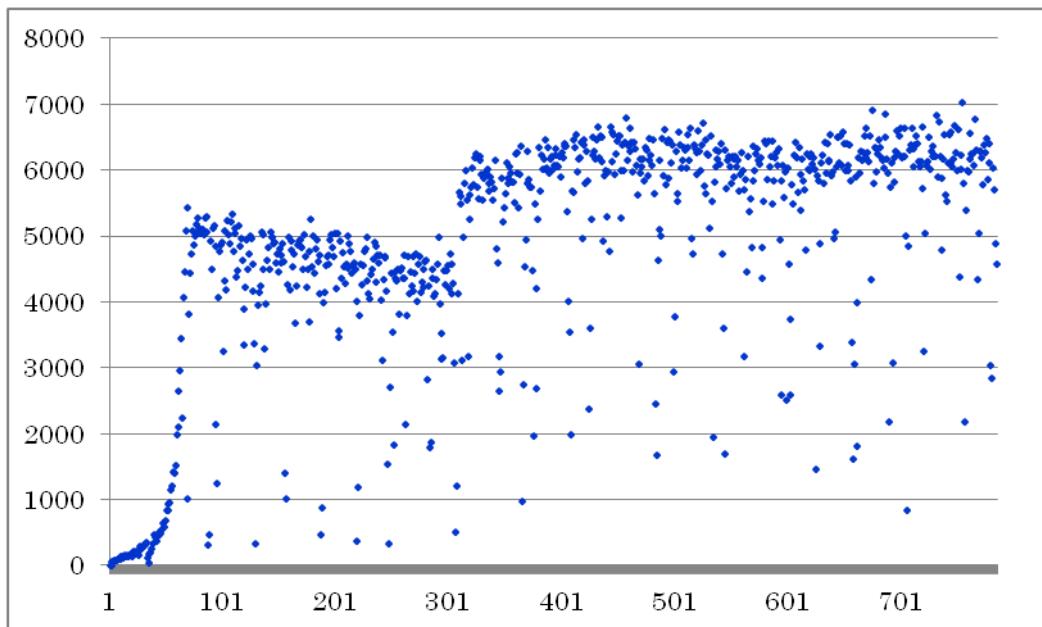
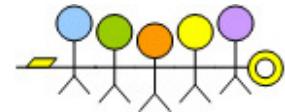
横坐标：时间(秒)

纵坐标：插入速度 (个/秒)



横坐标：时间(秒)

纵坐标：内存占用量(MB)(4G)



横坐标：时间(秒)

纵坐标：插入速度 (个/秒) (前 750)

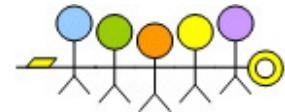
结论

插入速度和内存状态无关，随着数据量的增多，插入速度有下降的趋势，但是幅度很小；插入性能需要一段时间，才能提高到比较稳定的范围，震荡一直存在。

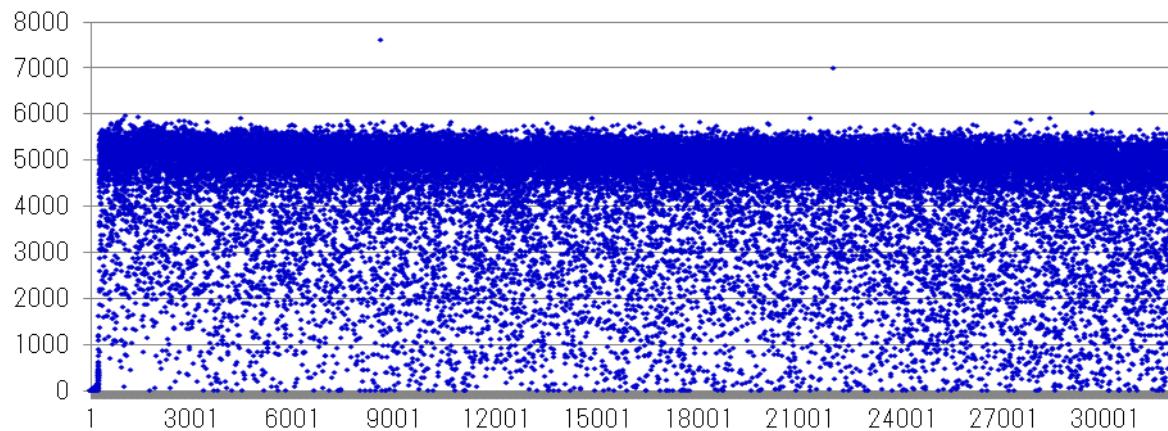
i5、2亿数据双索引插入（3亿基础上）性能测试

测试描述：

在3亿数据基础上单线程插入2亿条数据，有 双索引：number、name，内存10g

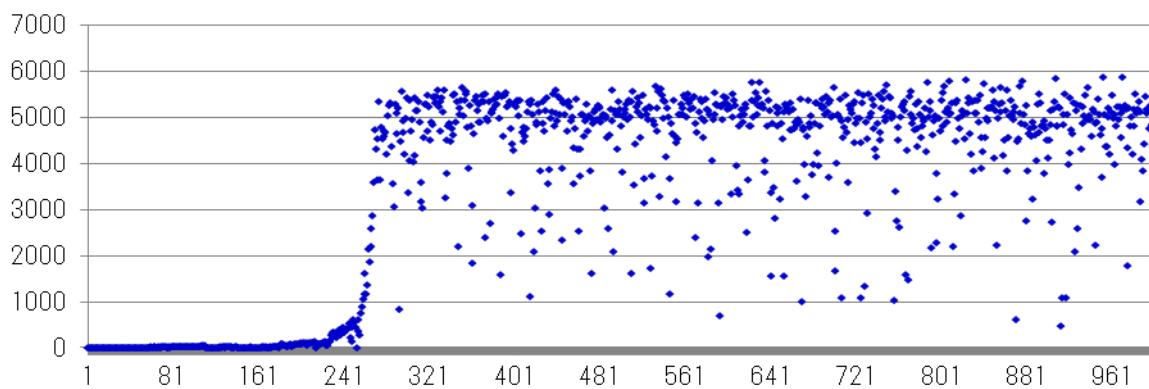


测试数据



横坐标：时间(秒)

纵坐标：插入速度 (个/秒)



横坐标：时间(秒)

纵坐标：插入速度 (个/秒) (前 1000)

结论

和 1 亿数据基础上的插入非常类似，速度略有下降，开始的提升过程需要更长的时间。

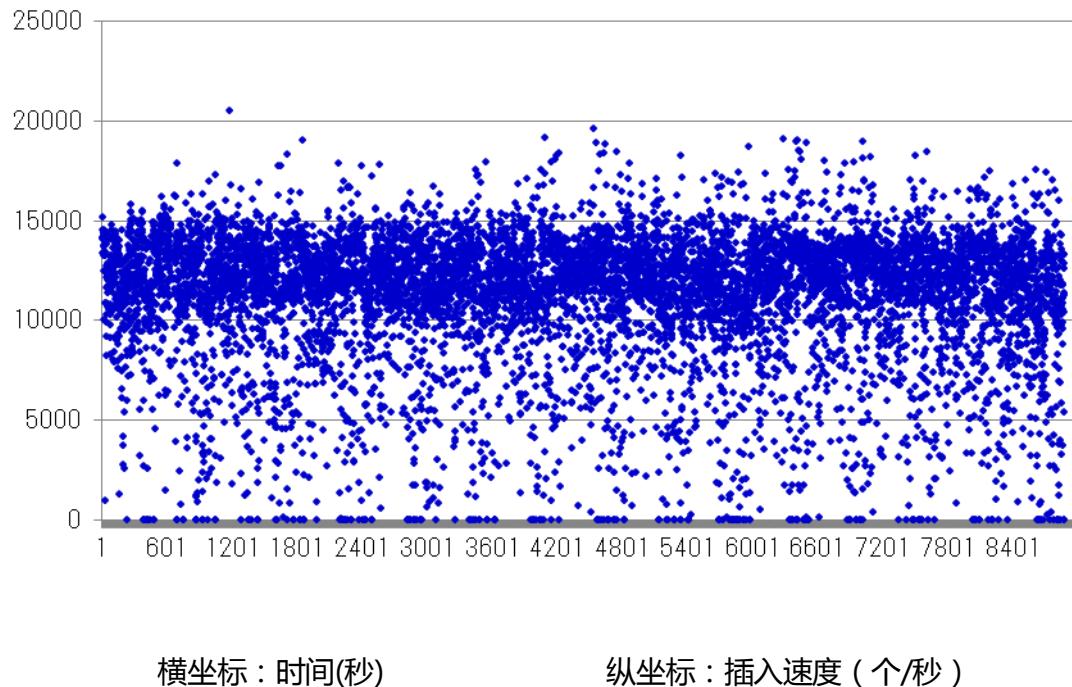
i6、Shard 插入性能测试

测试描述：

单线程，在 1 亿基础上插入 1000 万条数据，无索引，双服务器 shard。



测试数据



结论

速度很快，和单服务器的插入性能在同样的水平 ($>11K/s$)。

震荡原因分析：

在使用 Sharding 的时候，Mongodb 会对数据拆分搬迁，这个时候性能下降很厉害，但是我在实际观察中可以发现，在搬迁数据的时候每秒插入性能可能会下降一半。

其实我觉得能手动切分数据库就手动切分或者手动做历史库，不要依赖这种自动化的 Sharding，因为一开始数据就放到正确的位置比分隔再搬迁效率不知道高多少。

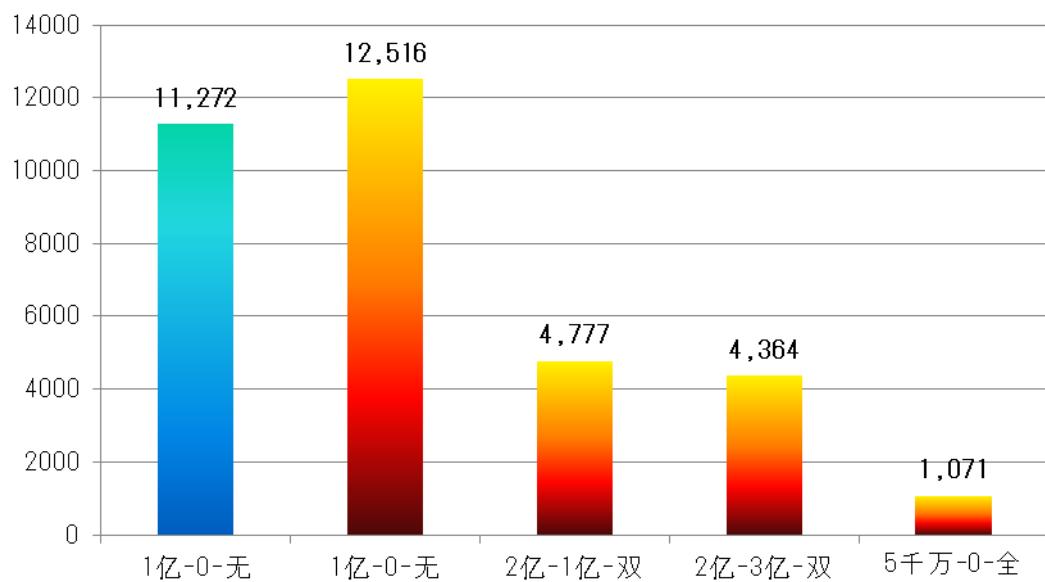


i7、五组大数据量插入性能对比

测试描述：

全部都是单线程插入，数据结构都是相同的，左 1 是 shard(双) ，其他都是单服务器

测试数据

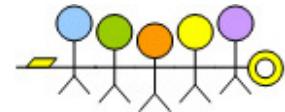


横坐标：(X-Y-Z) 纵坐标：插入速度 (个/秒)

X : 插入的数据量 ; Y : 数据库中已有数据量 ; Z : 索引个数

结论

索引数的增加会降低插入的性能，数据库中的总数据量增加也会降低插入性能，但是影响不大，Shard 的搭建会降低插入性能。

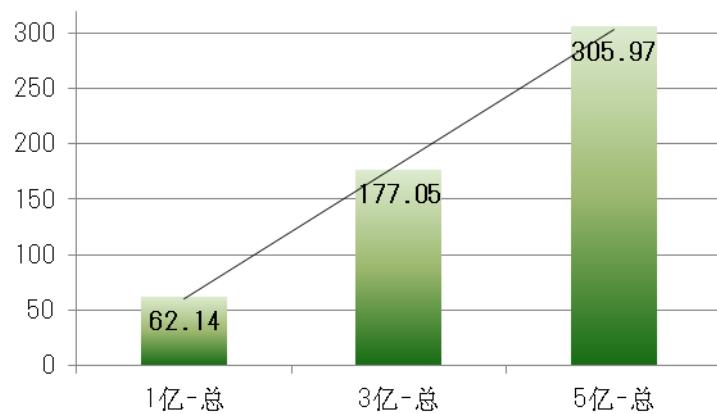


i8、不同数据量占用磁盘空间大小对比

测试描述：

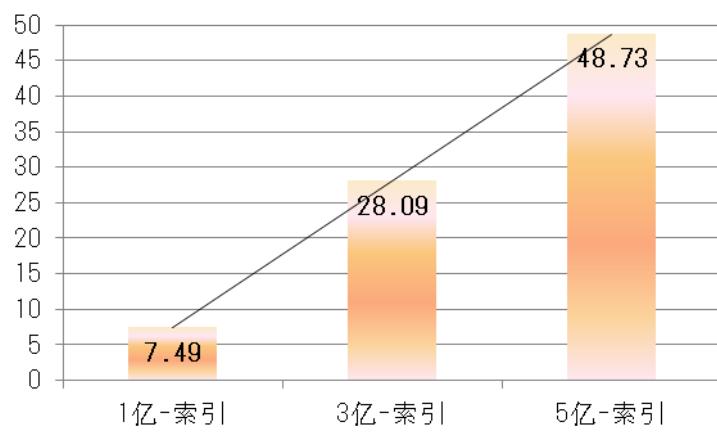
数据库中插入响应大小数据后，占用的磁盘空间大小，索引都只有三个：_id，number，name

测试数据



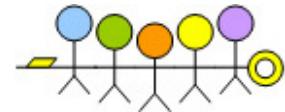
横坐标：数据量-总体

纵坐标：占用磁盘大小 (GB)



横坐标：数据量-索引

纵坐标：占用磁盘大小 (GB)



结论

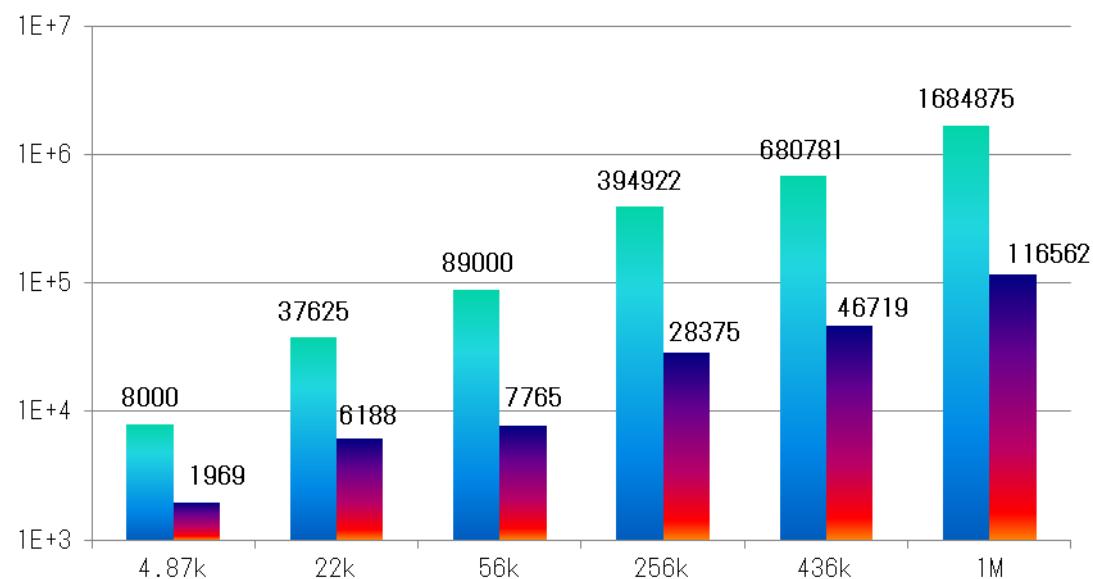
磁盘和索引占用的空间基本都随着数据量的增加而线性增长。

i9、GridFS 和二进制插入性能对比

测试描述：

单线程插入 1000 条， 数据字段 id, number, image。

测试数据



注：指数坐标 横坐标：图片大小 纵坐标：插入耗时 (毫秒)



二进制插入的 java 代码-范例

```
File ima = new File("436k.jpg");

FileInputStream fileIps = new FileInputStream(ima);

ByteArrayOutputStream bao = new ByteArrayOutputStream();

int imalength;

while ((imalength = fileIps.read()) != -1){

    bao.write(imalength);}

fileIps.close();

byte[] imabyte = bao.toByteArray();

bao.close();

document.put("imagByte",imabyte);
```

结论

无论图片大小，GridFS 存图片比二进制存储效率高一个数量级。

猜测

Java 的二进制存入代码可能有问题，单跑 put 之上的代码也要消耗同样多的时间。

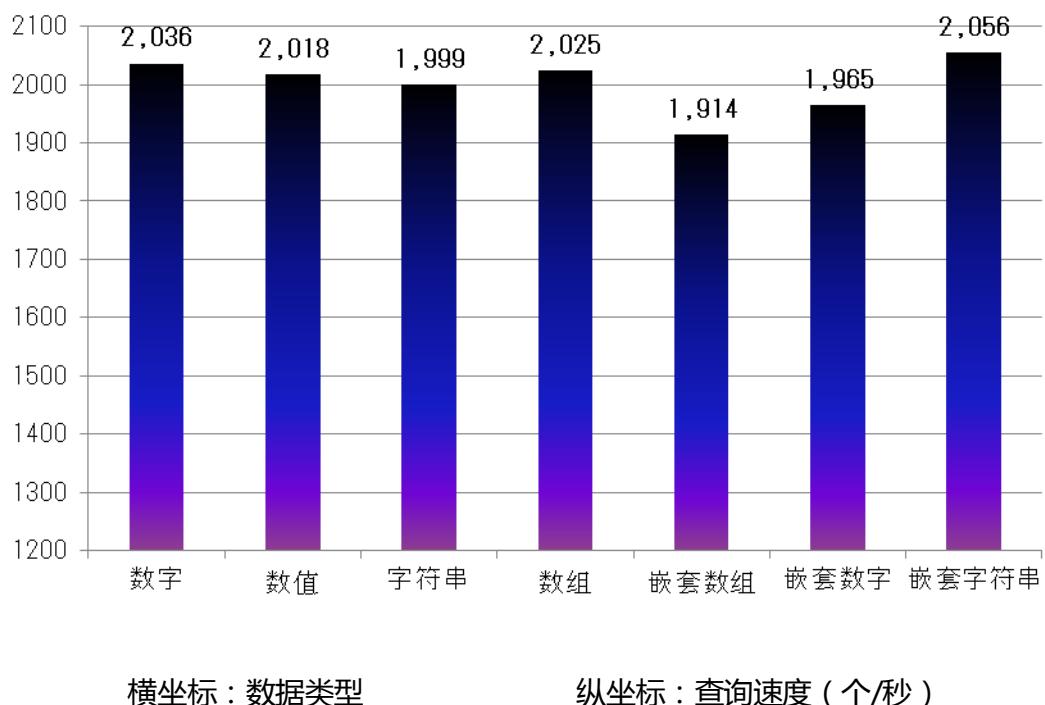


q1、针对不同数据类型查询的性能差别

测试描述：

五千万数据，全索引，单线程，数字列是查询 0-100 万随机，其他类型全口径随机，但只返回一条数据。

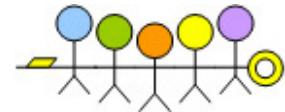
测试数据



结论

针对不同数据类型的查询，性能差别不大。

可以在大数据情况下只测试查询某一个数据项而能够代表其查询性能。

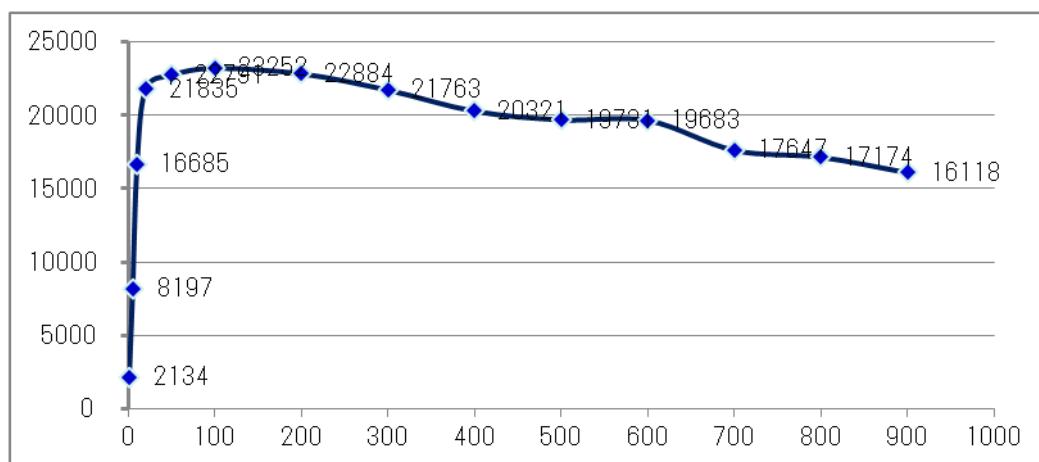


q2、并发数不同对于查询性能的影响

测试描述：

五千万数据，全索引，随机查询前 100 万数据

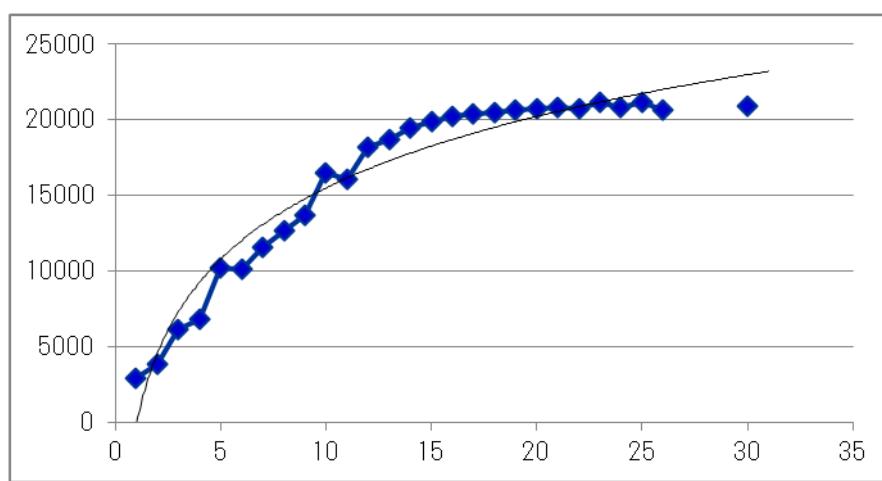
测试数据



线程数 1-900

横坐标：线程数

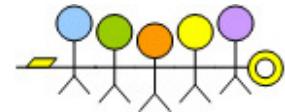
纵坐标：查询速度 (个/秒)



线程数 1-30

横坐标：线程数

纵坐标：查询速度 (个/秒)



结论

MongoDB 服务器的总体查询性能，随着并发数的增加，呈现某种类似抛物线的特点，过了最高点后会随着并发数增加而开始缓慢下降。

注：最大并发数 1013，在并发数达到 900 以后开始偶尔出现连接不上的问题。

猜测

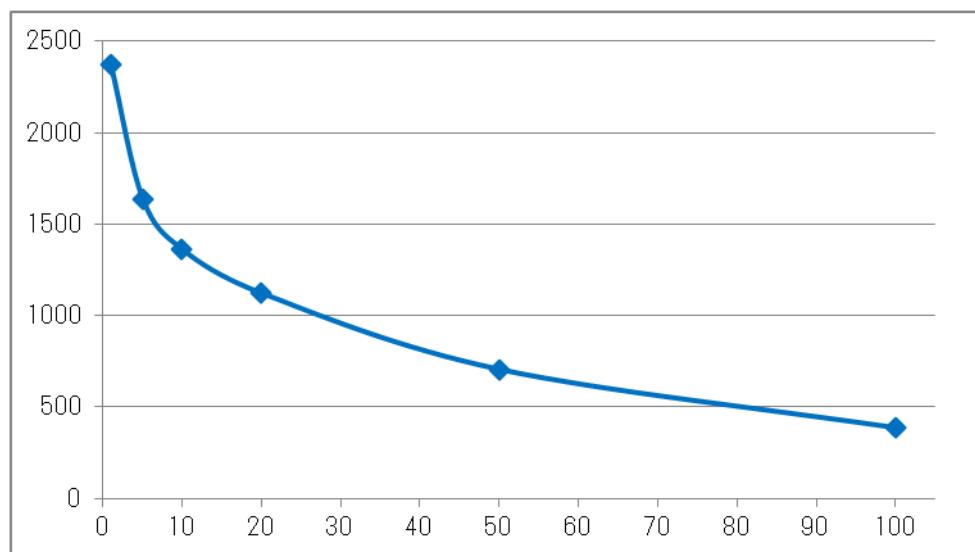
单线程查询速度比较慢的原因可能和脚本的性能有关。

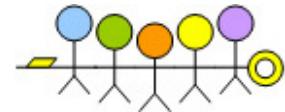
q3、Limit 对于查询性能的影响

测试描述：

五千万数据，全索引，单线程，查询 10 万次，随机查询 name，limit (x)

测试数据





横坐标 : limit(x) 的 X 值

纵坐标 : 查询速度 (个/秒)

结论

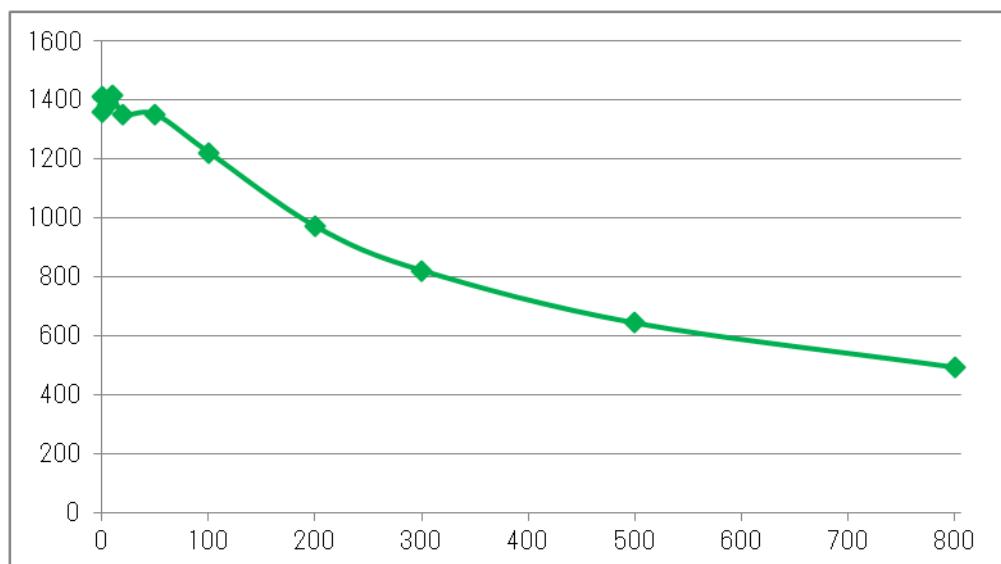
limit 值递增 , 每次查询的返回数据量递增 , 速度也随之下降。

q4、skip 对于查询性能的影响

测试描述 :

五千万数据 , 全索引 , 单线程 , 查询 10 万次 , 随机查询 name , skip(x).limit (10)

测试数据



横坐标 : skip(x).limit(10) 的 X 值

纵坐标 : 查询速度 (个/秒)

结论

skip 值递增 , 速度也随之下降 , 但是在 100 以内下降不明显。



q5、1亿数据随机全查性能测试

测试描述：

1亿数据，双索引 number , name , 20线程 , 10g 内存 , 随机全查全部 number , 即热数据 = 1亿条 , 远大于内存容量。截取数据为 mongostat

测试数据

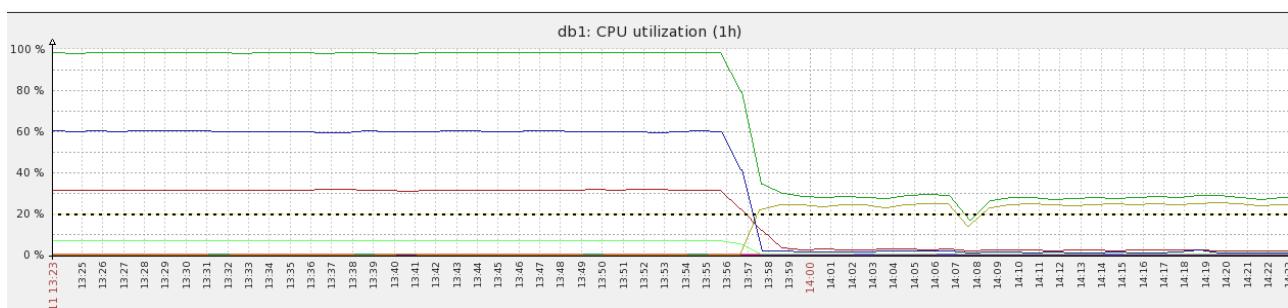


横坐标：时间(秒)

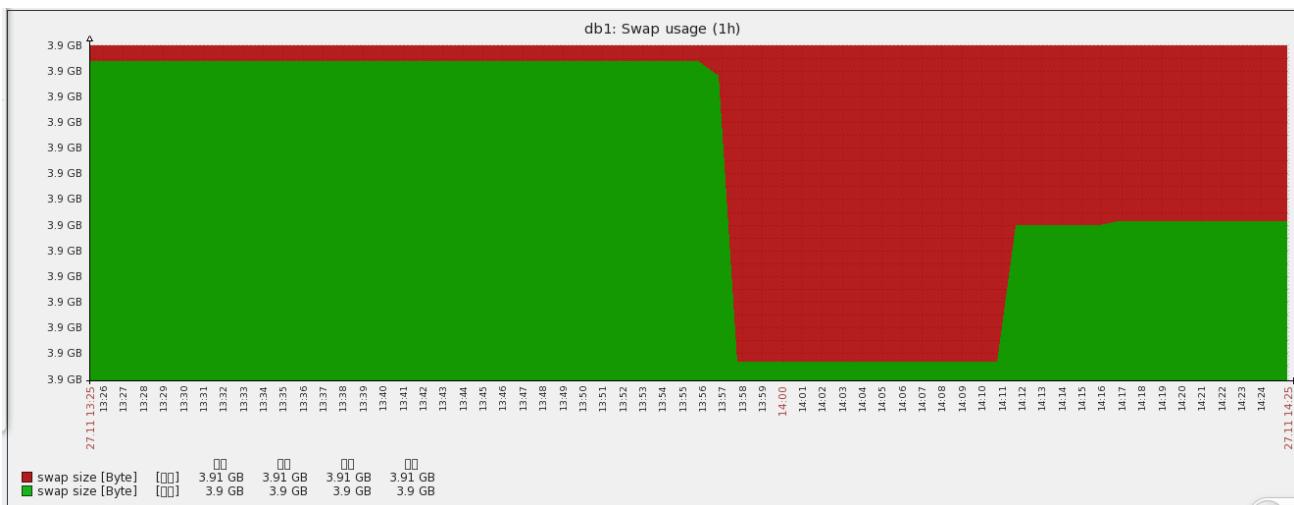
纵坐标：查询速度 (个/秒)

Zabbix 监测数据

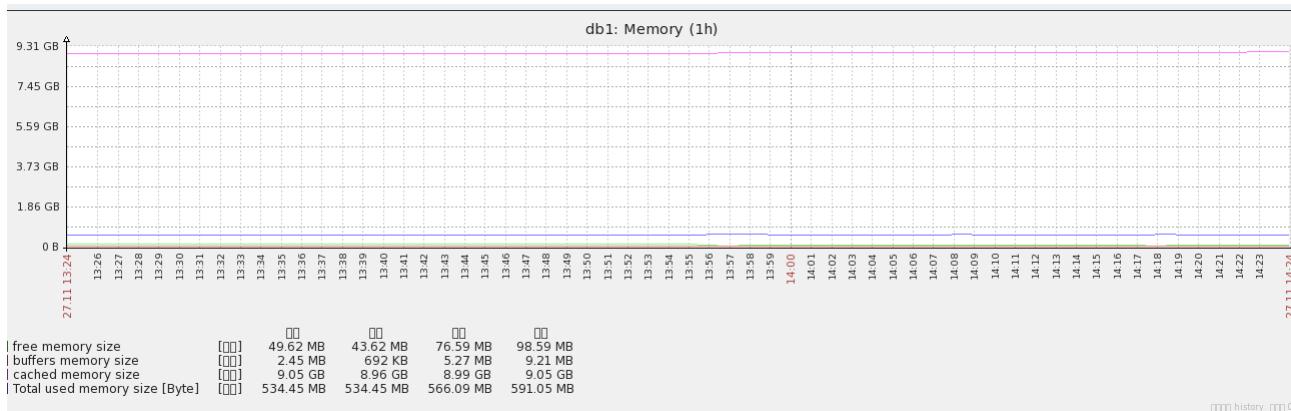
CPU 利用率很低 (后半段是该次测试的数据)



SWAP 使用频繁 (后半段是该次测试的数据)



内存长时间被吃满



结论

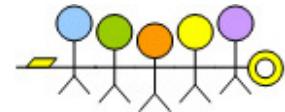
在热数据大幅度超过内存可容纳的量时，查询速度骤降到低位，并小幅度震荡。

q6、1亿数据随机查前1000万性能测试

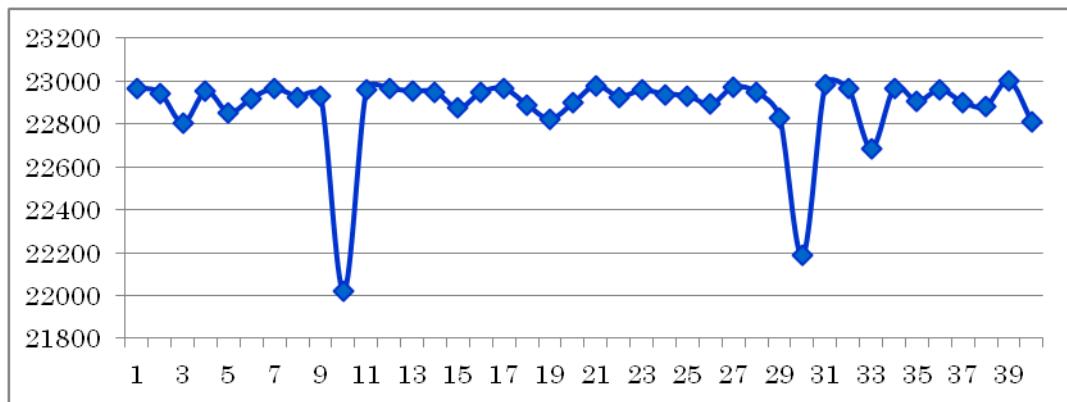
测试描述：

1亿数据，双索引| number , name , 20线程 , 10g 内存 , 随机全查全部 number , 即热数据 = 千

万条，小于内存容量。截取数据为 mongostat



测试数据

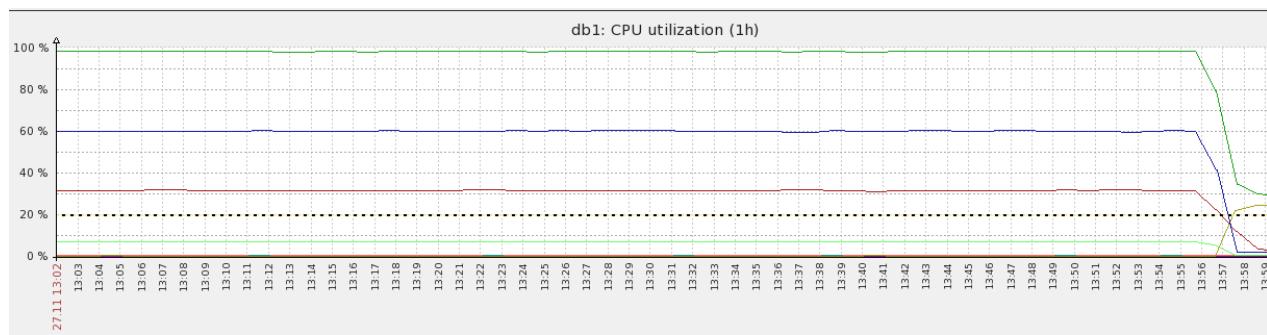


横坐标：时间(秒)

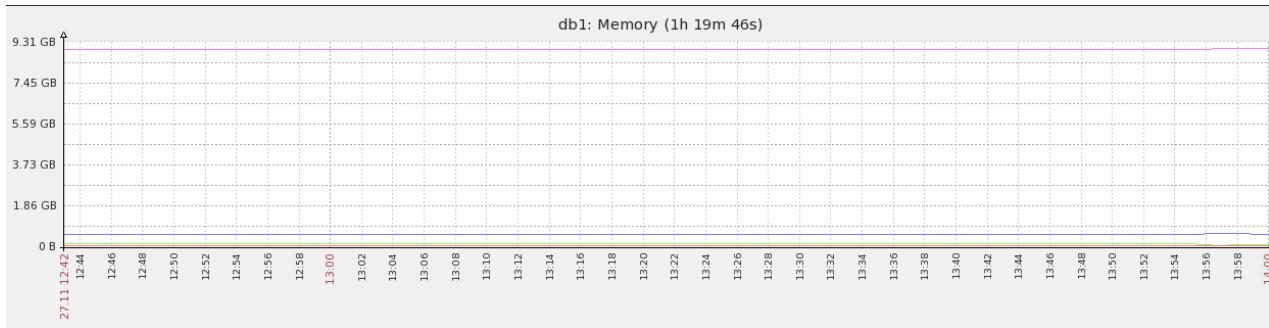
纵坐标：查询速度 (个/秒)

Zabbix 监测数据

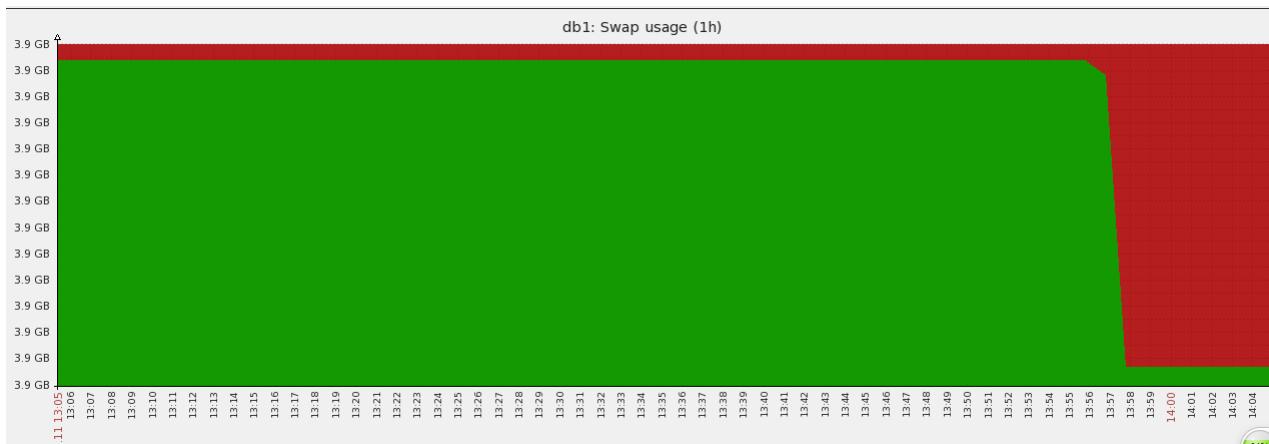
CPU 使用率处于高位



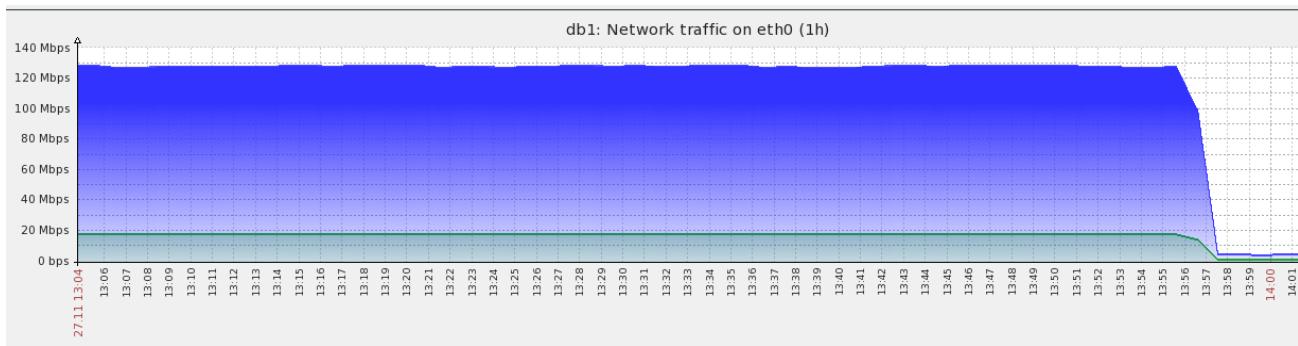
内存吃满



SWAP 使用率很低

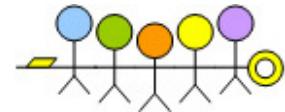


I/O 处于高位



结论

在热数据可全部装入内存时，查询速度在高位小幅度震荡。

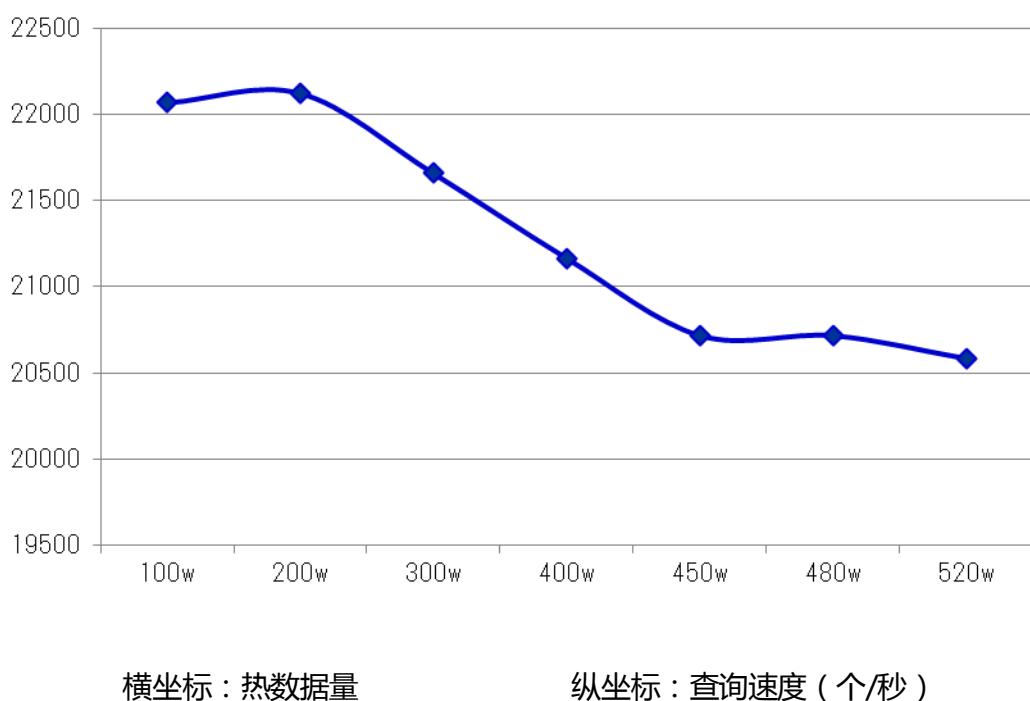


q7、热数据量大小 (< 内存大小) 对于查询性能的影响

测试描述：

五千万数据，全索引，20 线程，4G 内存，随机查询前 X 万数据，查询截取 faults 长时间稳定为 0 的情况，即：热数据全部装入内存。

测试数据



结论

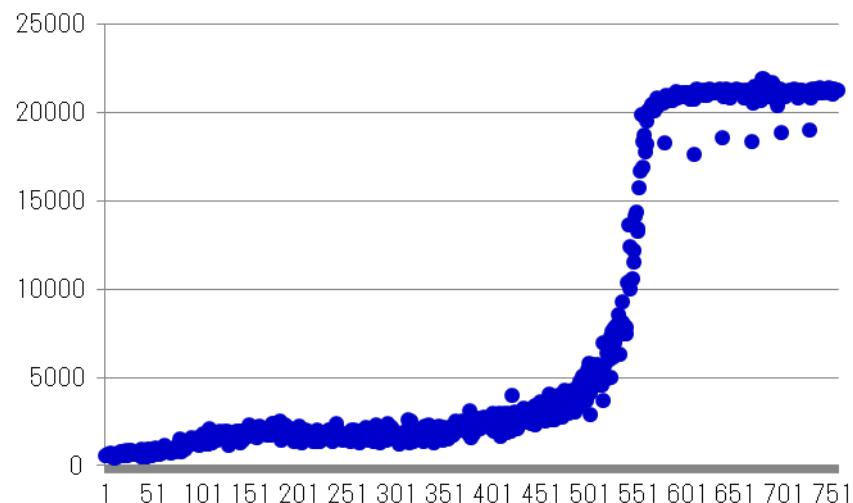
热数据量增加，查询性能缓慢下降。另：热数据量增加到内存无法装下时，性能骤降 2 个数量级。

q8、热数据进入内存过程对于查询性能的影响

测试描述：

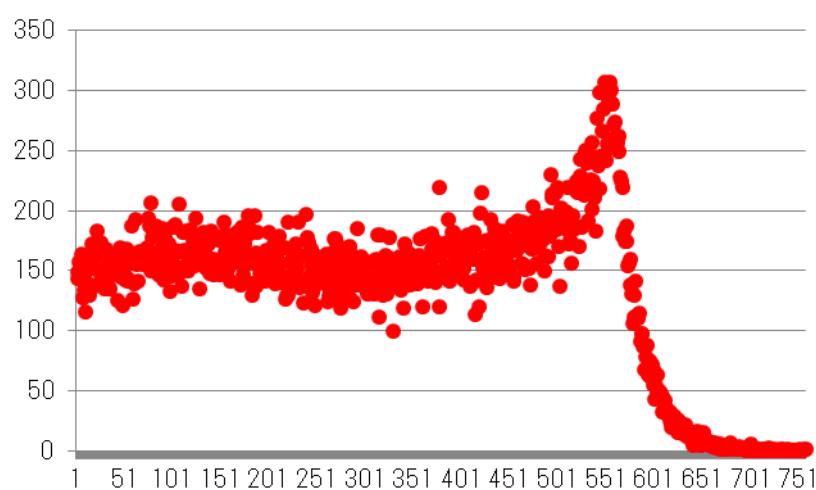
五千万数据，全索引，4G 内，20 线程，随机查询前 450 万数据

测试数据



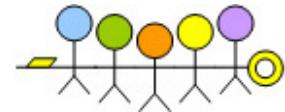
横坐标：时间(秒)

纵坐标：查询速度 (个/秒)



横坐标：时间(秒)

纵坐标：mongostat-faults 数量 (查询未命中数量)



结论

随着热数据逐步进入内存，查询速度缓慢上升，在热数据全部进入内存后，Faults 下降到 0，查询速度迅速攀升到峰值，并稳定维持。

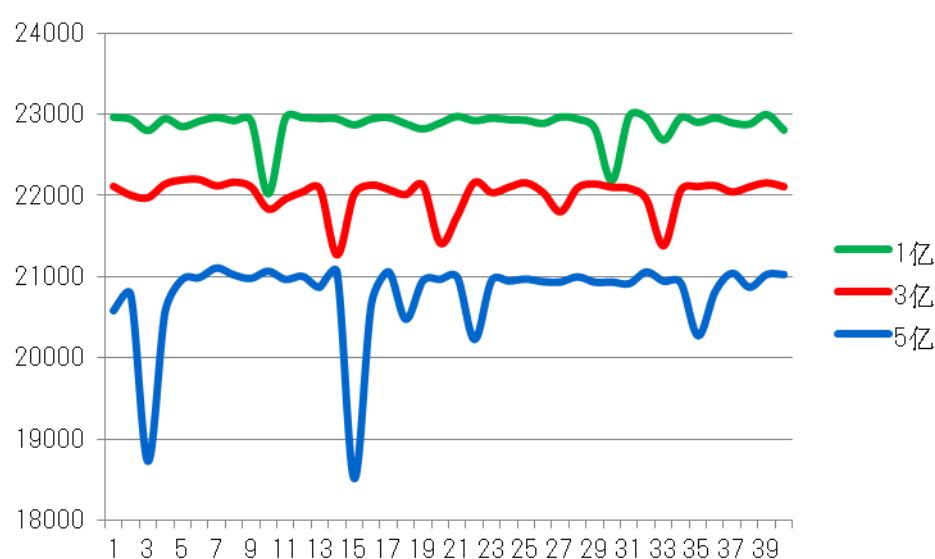
注：热数据进入内存的时间，可能不是线性的.即 1000 万热数据进入内存的时间可能远超过 100 万热数据进入内存的时间的 10 倍。

q9、1-3-5 亿数据量查询前 1000 万性能对比

测试描述：

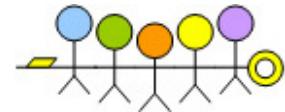
10G 内存，20 线程，随机查询前 1000 万数据（热数据全部能进入内存），Faults 长时间稳定为 0 后截取数据。

测试数据



横坐标：时间(秒)

纵坐标：查询速度 (个/秒)



结论

相同热数据量（< 内存），查询性能随着总数据量的增加而下降。

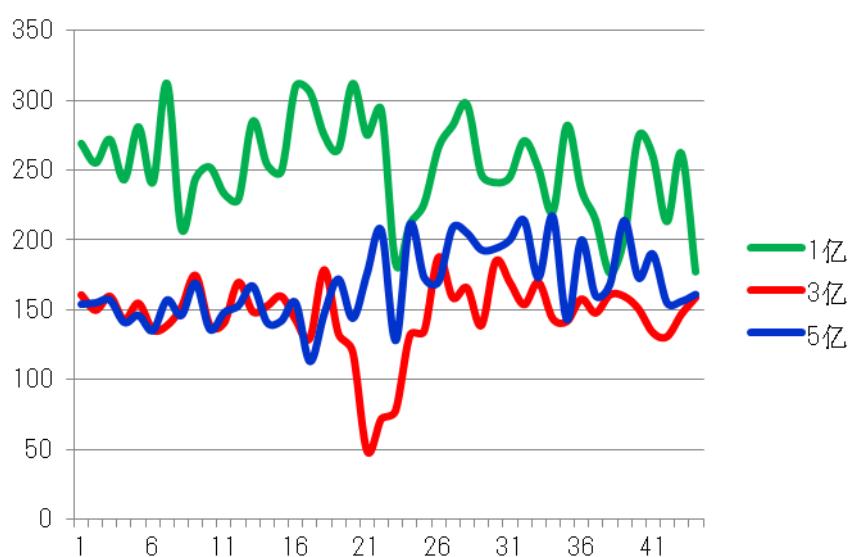
注：只截取了 40 个点，所以观测的震荡比较明显，长时间的数据总体是稳定的。

q10、1-3-5 亿数据量查询前 1 亿性能对比

测试描述：

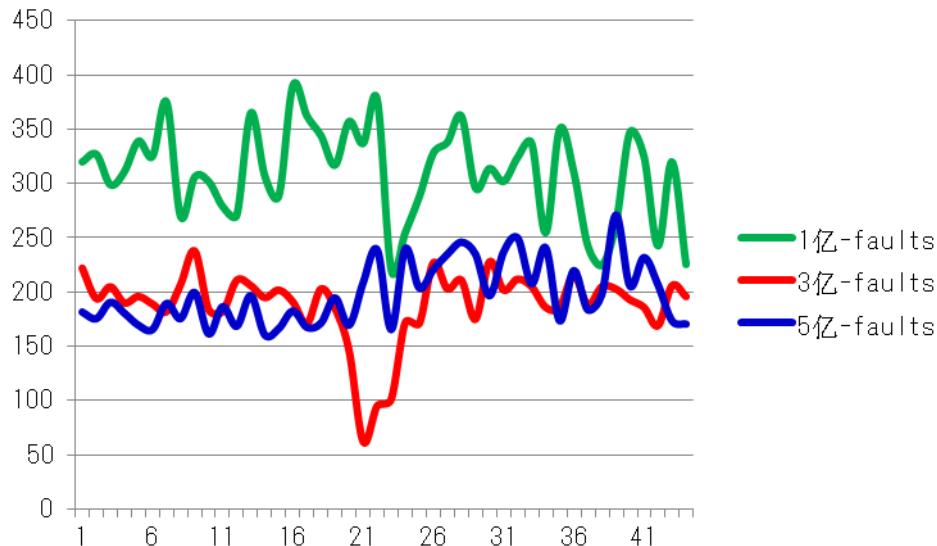
10G 内存（热数据量远大于内存容量），20 线程，随机查询前 1 亿数据，内存 res 基本稳定后截取数据。

测试数据



横坐标：时间(秒)

纵坐标：查询速度 (个/秒)



横坐标：时间(秒)

纵坐标：mongostat-faults 数量 (查询未命中数量)

结论

热数据量增加到远大于内存后，总数据量增加还是会降低查询速度，但是影响非常不明显，震荡比例也变得剧烈。

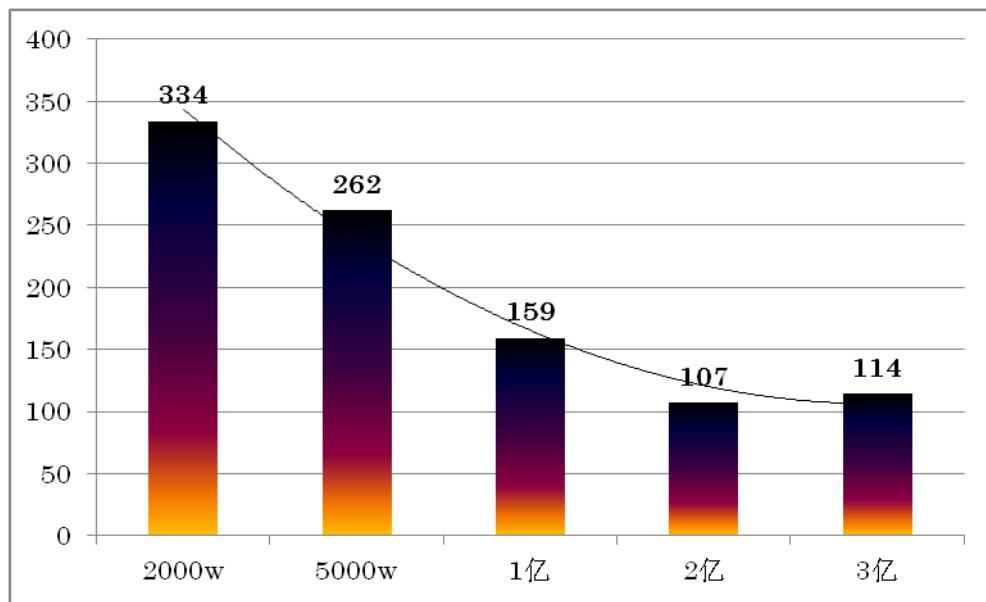
Faults 和查询速度基本上同步

q11、1-3-5 亿数据不同热数据量查询性能对比

测试描述：

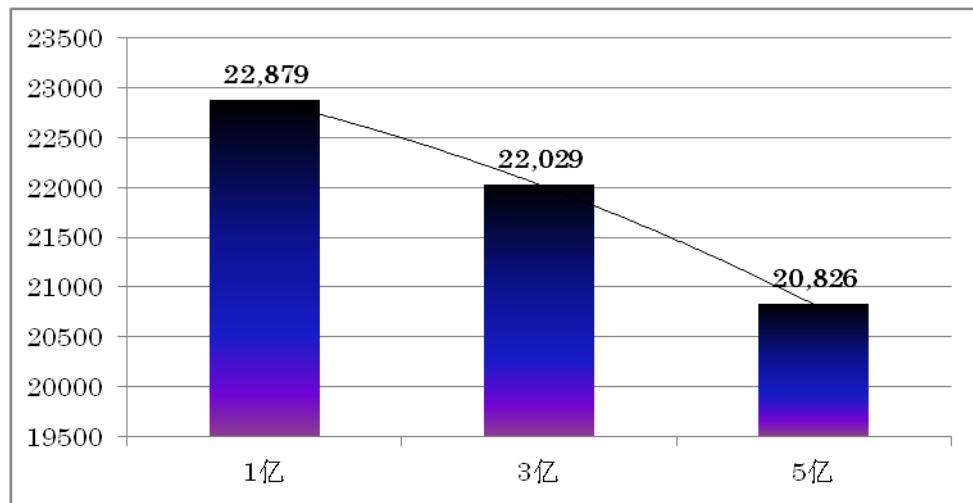
双索引，10g 内存，20 线程，随机查询 number

测试数据



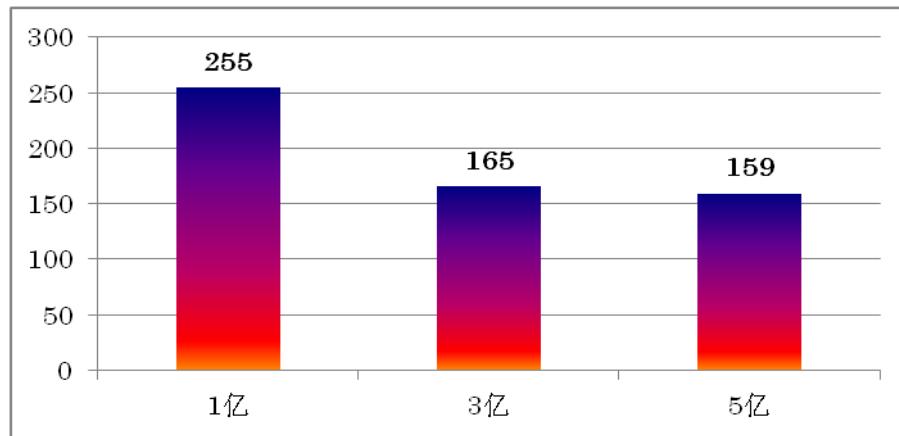
横坐标：5亿数据中随机查询前 X 条数据

纵坐标：查询速度（个/秒）

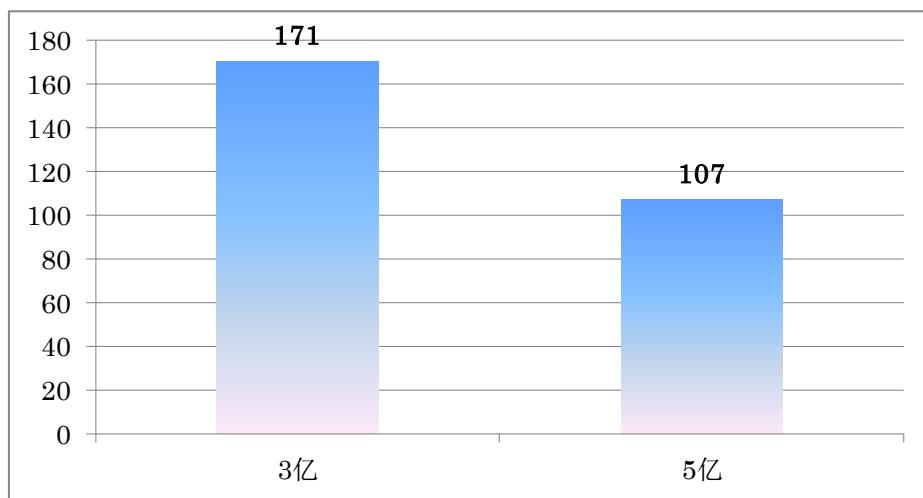


横坐标：X 数据中随机查询前 1000 万

纵坐标：查询速度（个/秒）



横坐标：X 数据中随机查询前 1 亿 纵坐标：查询速度（个/秒）



横坐标：X 数据中随机查询前 2 亿 纵坐标：查询速度（个/秒）

结论

总数据量和热数据量的大小都会对查询速度产生影响，热数据量能否全部进入内存对于性能影响

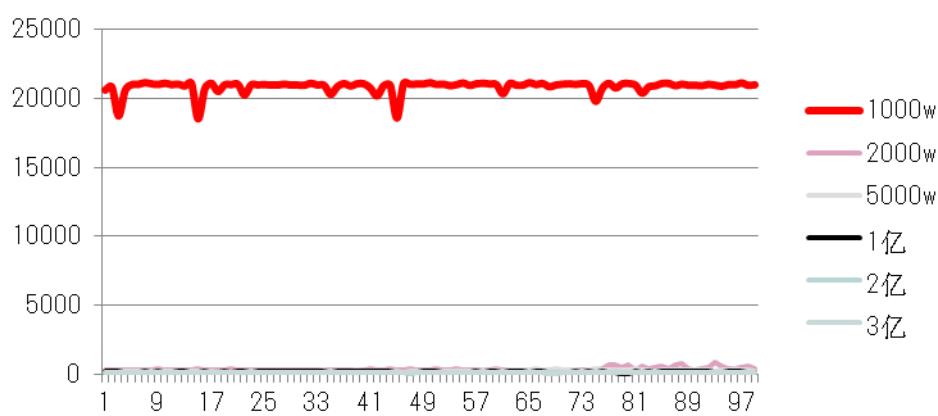
巨大，差两个数量级，总数据量的增加和超过内存大小的热数据量的增加对于查询性能的影响不大。

q12、5亿数据量查询不同数据量性能对比

测试描述：

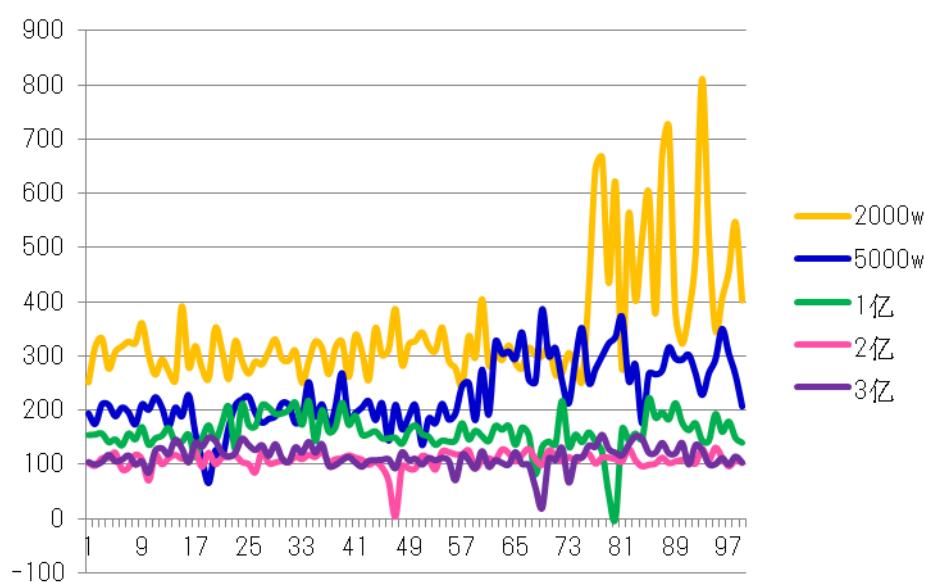
10G 内存，20 线程，5 亿数据量，随机查询不同数据量，内存 res 基本稳定后截取数据

测试数据



横坐标：时间(秒)

纵坐标：查询速度 (个/秒)



热数据量大于内存部分详细：横坐标：时间(秒)

纵坐标：查询速度 (个/秒)



结论

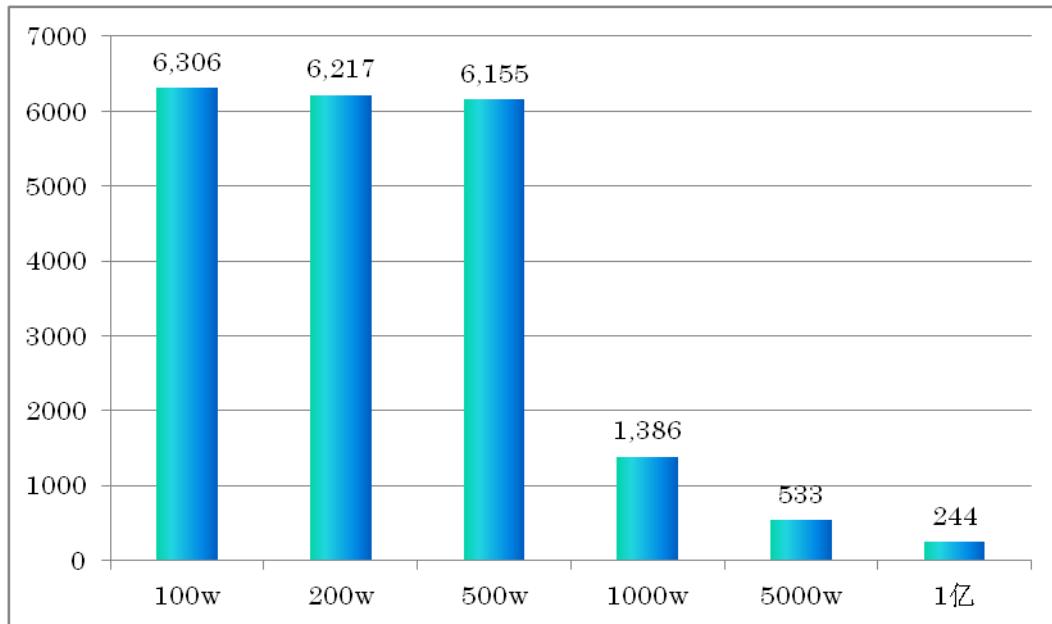
热数据能否全部装入内存对于查询速度影响巨大。热数据量超过内存大小时，热数据量的增加会降低查询性能，但影响变得很小。

q13、Shard(双机) 查询性能测试

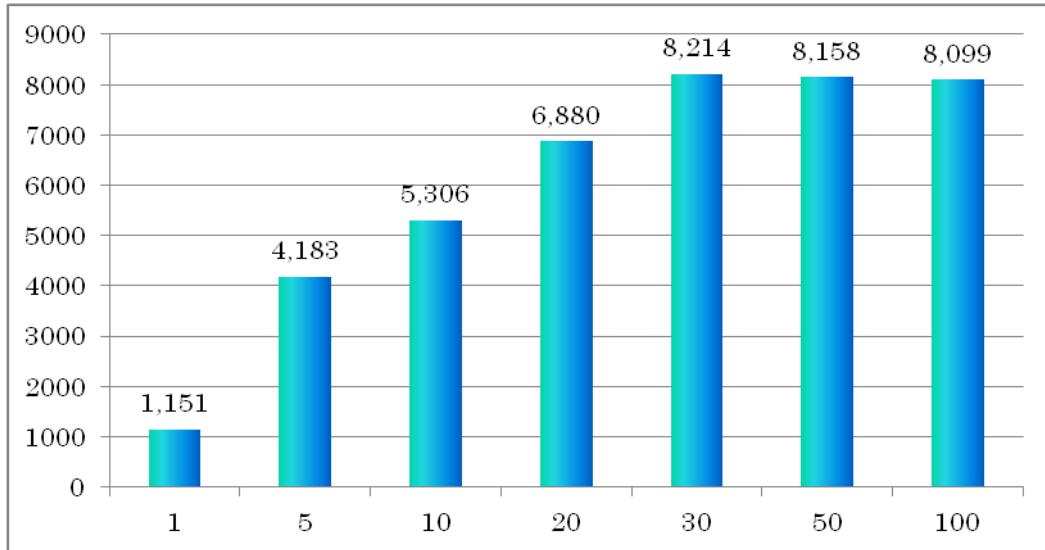
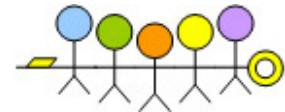
测试描述：

1亿数据量，单索引| number

测试数据

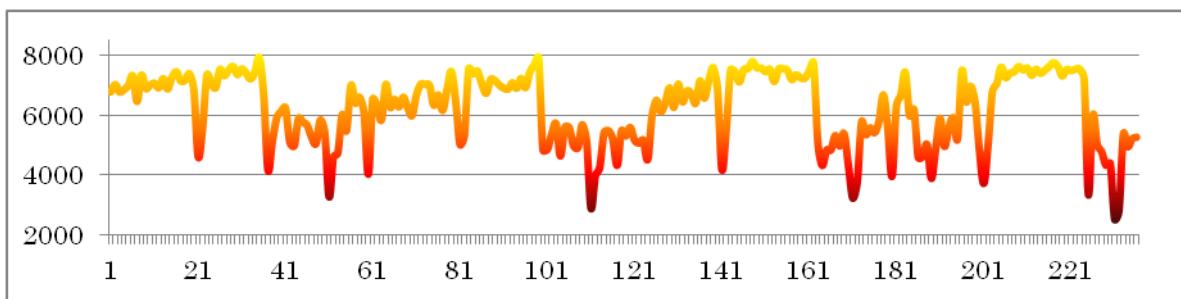


20 线程 : 横坐标 : 热数据量 纵坐标 : 查询速度 (个/秒)



横坐标：线程数

纵坐标：查询速度 (个/秒)



横坐标：时间

纵坐标：查询速度 (个/秒)

结论

查询的最快速度在 8,000 附近，相比单服务器 (2.3 万) 的查询速度有明显的下降，其他方面的性能和单服务器的查询性能类似。震荡很有周期性。

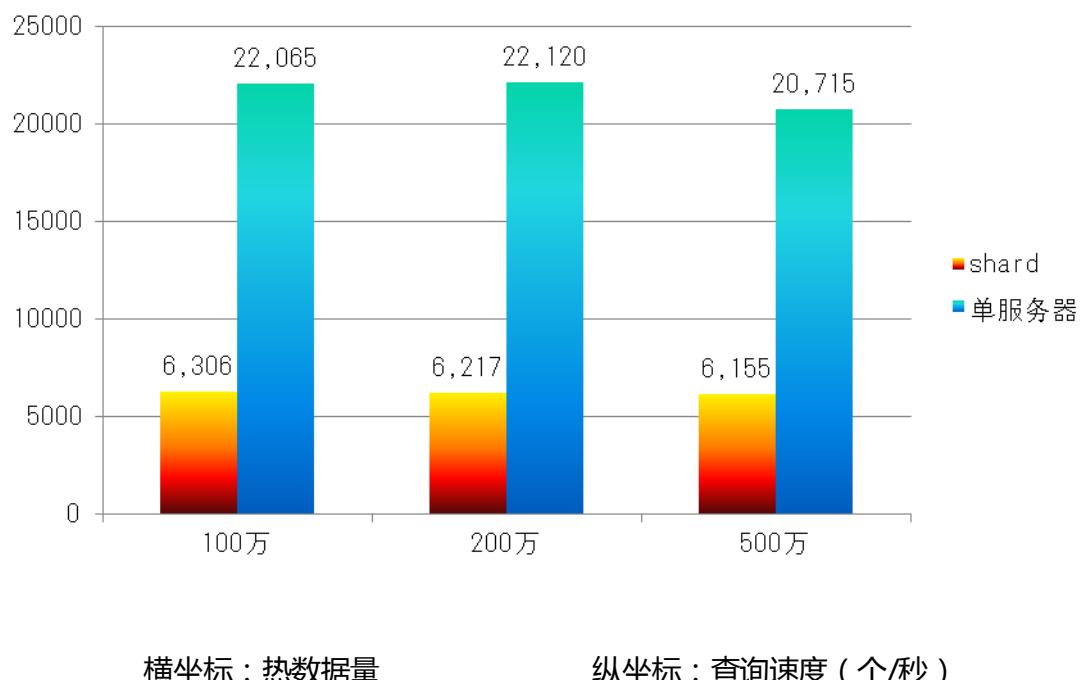


q14、Shard(双机) 和单服务器的查询性能对比

测试描述：

20 线程，随机查询 number , Shard : 1亿数据，单服务器： 5 千万数据。

测试数据



横坐标：热数据量

纵坐标：查询速度（个/秒）

结论

Shard 查询速度相比单服务器明显下降。

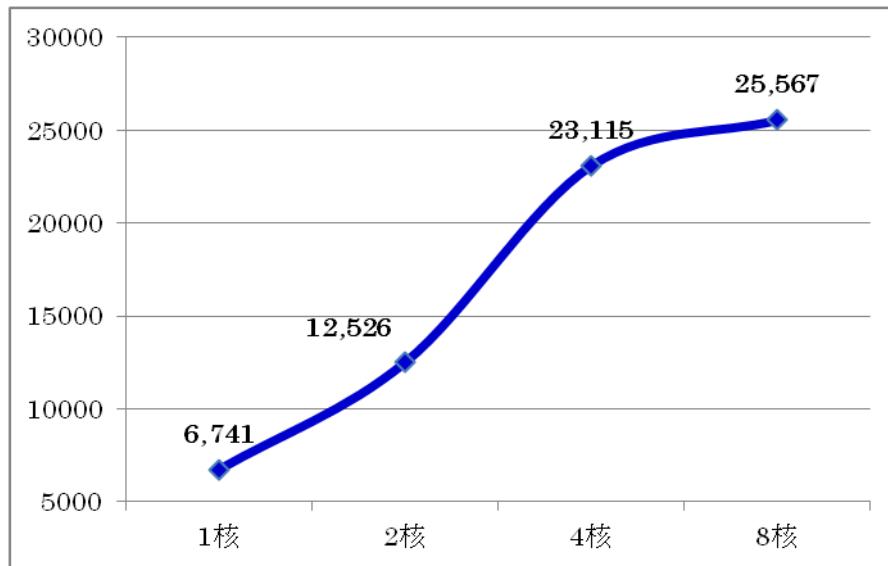
q15、CPU 与查询性能的关系

测试描述：

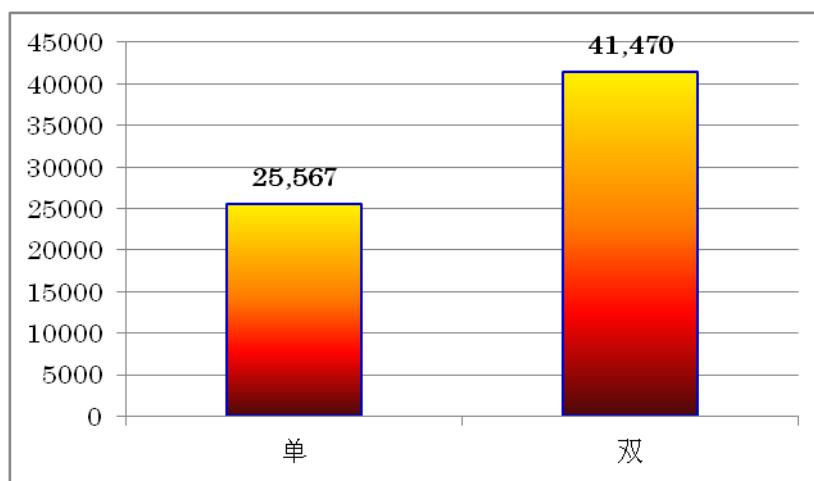
5 亿数据，20 线程，查询前 100 万。



测试数据



横坐标 : CPU 数量 纵坐标 : 查询速度 (个/秒)



横坐标 : 加压机数量 纵坐标 : 查询速度 (个/秒)

结论

查询性能随着 CPU 数量的增加而线性增加。

另：

4 核之前，速度随着 CPU 的增加线性增加，8 核降低由于加压机功率不够。

8 核， “单” 的限制条件加压机的 CPU 达到上限。

o1、不同情况建立索引的对比

现象描述：

CPU 爆满、内存吃满、locked 锁死

索引项	情况	耗时
Price	单服务器、1亿数据、10g 内存	约 54 分钟
Price	单服务器、3亿数据、10g 内存	约 5 小时
number	Shard-双机、1亿数据、4g 内存	约 19 分钟

结论

建立索引的时间随着数据量的增加呈现快速增长，而且在建立索引时，服务器什么都做不了，完全锁死。

注：如果在大数据量上建立索引，要有足够的时间

o2、repairDatabase 的现象

现象描述

repairDatabase 执行 3 小时左右，期间 CPU 爆满、内存吃满、locked 锁死



操作描述	数据库大小
插入 1 亿无索引数据	69G
插入 1 千万数据后	79G
把这 1 千万数据删除	79G
执行 repairDatabase	71G

结论

数据库空间会动态增长但是不会动态减少，

另：

MongoDB 分配空间的原则是不够用就动态增加一个数据块， 数据块的大小最开始是 16M，然后 32M，然后 64M，依此翻倍增加到 2G 以后，数据库大小不再增加，每次都是分配 2G 大小的数据块。

测试结论

插入

- 1 内存状态(装满与否)对于插入速度没有影响
- 2 插入速度随着数据量的增加而缓慢下降
- 3 批量插入的过程中震荡一直存在，且幅度较大



- 4 有索引的批量插入在开始时插入速度有一个从慢到快的过程，这一过程随着数据量的增加而逐渐延长
- 5 索引的建立会明显降低插入速度，索引数越多插入速度越慢
- 6 磁盘和索引占用的磁盘空间基本都随着数据量的增加而线性增长
- 7 无论图片大小，GridFS 存图片比二进制存储效率高

查询

- 8 针对不同数据类型的查询，性能几乎没有差别
- 9 线程数 20 之前，查询性能随线程数增加线性增加，在线程数超过 100 后，查询性能随着线程数增加缓慢下降，在 1013 开始报错
- 10 limit 值递增，每次查询的返回数据量递增，速度也随之下降
- 11 skip 值递增，速度也随之下降，但是在 100 以内下降不明显
- 12 热数据全部装入内存时，查询速度在高位小幅度震荡
- 13 热数据量超过内存大小时，查询速度骤降两个数量级
- 14 查询性能随着热数据量（大于或者小于内存的两侧）的增加而缓慢下降
- 15 查询性能随着总数据量的增加而缓慢下降
- 16 Shard 查询速度相比单服务器明显下降
- 17 查询性能随着 CPU 数量的增加而线性增加

没有解决的问题

1. 连接数问题，con 无法突破 1013
2. locked 比例过高问题（建索引，批量插入等行为）
3. 内存大小和理想热数据量的函数关系
4. GridFS 存储的具体机制
5. 性能震荡
6. 不同语言驱动性能不同
7. 索引数量与插入速度的函数关系
8. 安全插入相关性能测试
9. 排序、\$gte、\$lte 的性能
10. 【错误 1067】MongoDB 异常关闭，无法正常启动，需删除 lock 文件
11. Mongostat 的内存 res 和 Zabbix 的内存不一致