# HBase基准性能测试报告

## 前言

本次测试主要评估HBase的整体性能，量化当前HBase的性能指标，对各种场景下HBase性能表现进行评估，为业务应用提供参考。

本报告主要介绍此次测试的基本条件、HBase在各种测试场景下的性能指标（主要包括单次请求平均延迟和系统吞吐量）以及对应的资源利用情况，并对各种测试结果进行分析。

## 测试环境

### 硬件配置

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| IP | CPU型号 | 核心数 | 内存(GB) | 磁盘(TB) | 以太网(Mbps) |
| 192.168.2.101 | Genuine Intel(R) CPU @ 2.20GHz | 24 | 125 | 12 | 1000 |
| 192.168.2.102 | Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz | 16 | 125 | 3.4 | 1000 |
| 192.168.2.103 | Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz | 40 | 109 | 3.4 | 1000 |

### 软件配置

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| IP | 操作系统 | JVM | Hadoop | HBase | 测试工具 |
| 192.168.2.101 | CentOS Linux release 7.5.1804 (Core) | openjdk version "1.8.0\_252" | 2.7.7 | 1.2.5 | YCSB 0.17.0 |
| 192.168.2.102 |
| 192.168.2.103 |

### HBase核心配置

|  |  |
| --- | --- |
| 配置项 | 值 |
| Xmx | 80g |
| XX | UseG1GC |
| hbase.regionserver.global.memstore.size | 0.4 |
| hfile.block.cache.size | 0.4 |
| hbase.regionserver.handler.count | 500 |

**其他说明**

本次测试环境机器硬件配置不同，JVM并不是HotSpot JVM，而且HBase环境本身不纯净，以上因素都会对测试结果造成无法评估的影响。因此，本次测试结果仅供参考，不建议作为评估生产环境资源的依据。

## 测试方案

### 原则

1. 可测试
2. 可重复
3. 可对比

### 二、指标

|  |  |  |  |
| --- | --- | --- | --- |
| CPU | 带宽 | 网络延迟 | 吞吐量 |
| 包括CPU使用率，CPU负载，CPU中断等 | 包括网络IO和磁盘IO | 开始执行一个操作到操作完成中间的耗时 | 每秒钟执行操作的次数 |

### 三、变量

|  |  |  |
| --- | --- | --- |
| 数据量 | 并发数 | 压缩算法 |
| 测试不同数据量及大小（包括key、field、value、region）下的读写性能差异 | 测试不同并发下读写性能的差异 | 测试不同压缩算法和编码算法下，读写性能的差异 |

### 四、场景

|  |  |
| --- | --- |
| 场景 | 说明 |
| 单条查询 |  |
| 单条插入 |  |
| 批量查询 | 每个操作查询1000条数据 |
| 批量插入 | 每个操作插入1000条数据 |
| scan查询 | 每个操作扫描1000条数据 |
| 插入为主 | 5%的读操作，95%的写操作 |
| 查询为主 | 95%的读操作，5%的写操作 |
| 读写平衡 | 50%的读操作，50%的写操作 |

### 流程

1. 在3个客户端服务器上部署YCSB测试程序，按配置向集群中load数据；
2. 分别按照预先准备好的负载配置文件，运行YCSB测试程序；
3. 统计YCSB程序产生的结果；
4. 根据结果生成对应图表；
5. 结合zabbix的系统监控报告，对结果进行分析；
6. 重复以上步骤，直至所有测试场景测试完毕。

## 测试结果

### 每个表不同预分区数量下的性能对比

#### 方案

|  |  |
| --- | --- |
| 配置项 | 说明 |
| 数据条数 | 2000万 |
| 读/写次数 | 每客户端各执行666万6667次 |
| 预分区数量 | 自变量 |
| 列族 | 1个 |
| rowkey大小 | 14字节 |
| 字段数量 | 10个 |
| 字段大小 | 10字节 |
| 请求分布模型 | zipfian |
| 客户端数量 | 3个 |
| 线程数 | 50 |
| 统计指标 | 吞吐量取三者平均 |
| 延迟取三者平均 |

#### 测试结果

1. **读内存**

图表 1

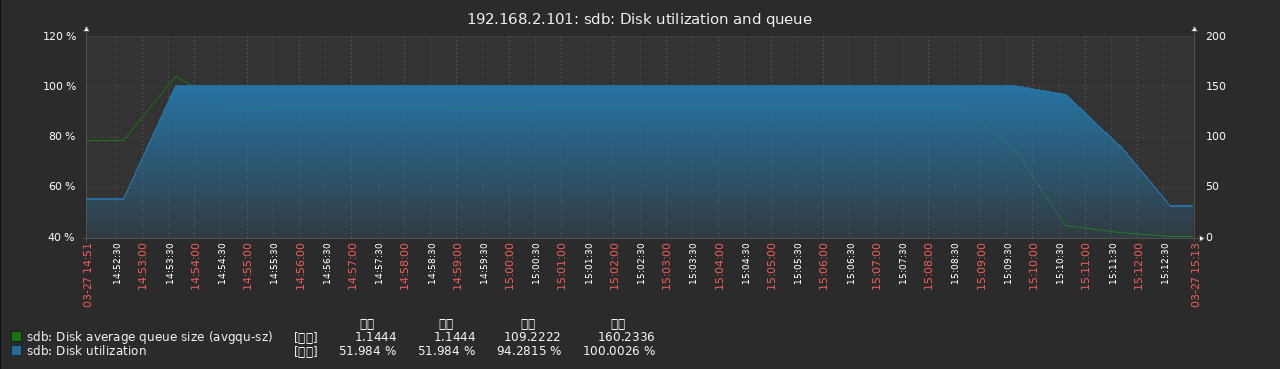
根据上图可得：

1. 在不进行预分区时，吞吐量和延迟都不是很理想；
2. 在小数据量下，不同预分区数量的性能差距并不明显；
3. **读磁盘**

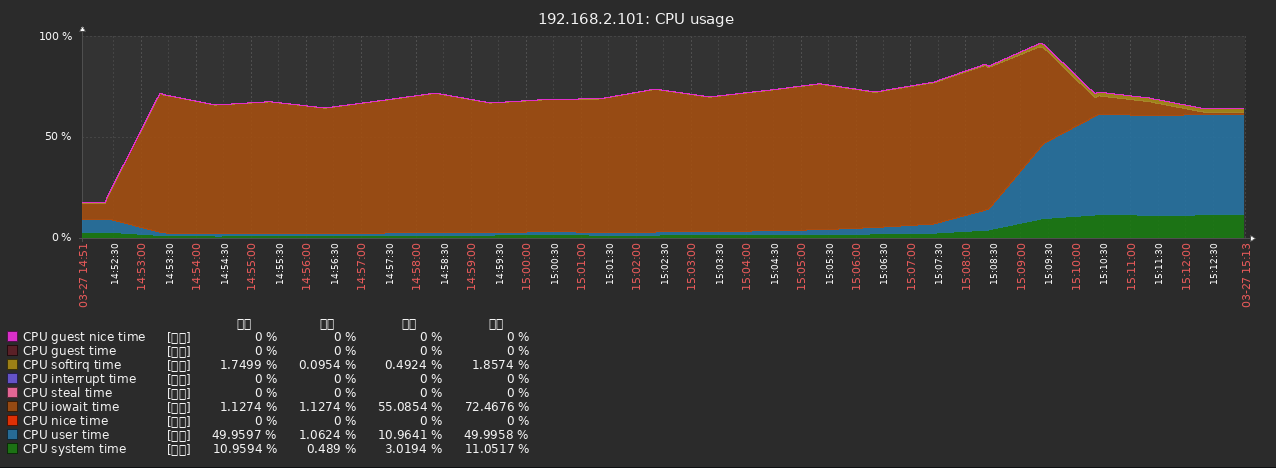
图表 2

根据上图可得：

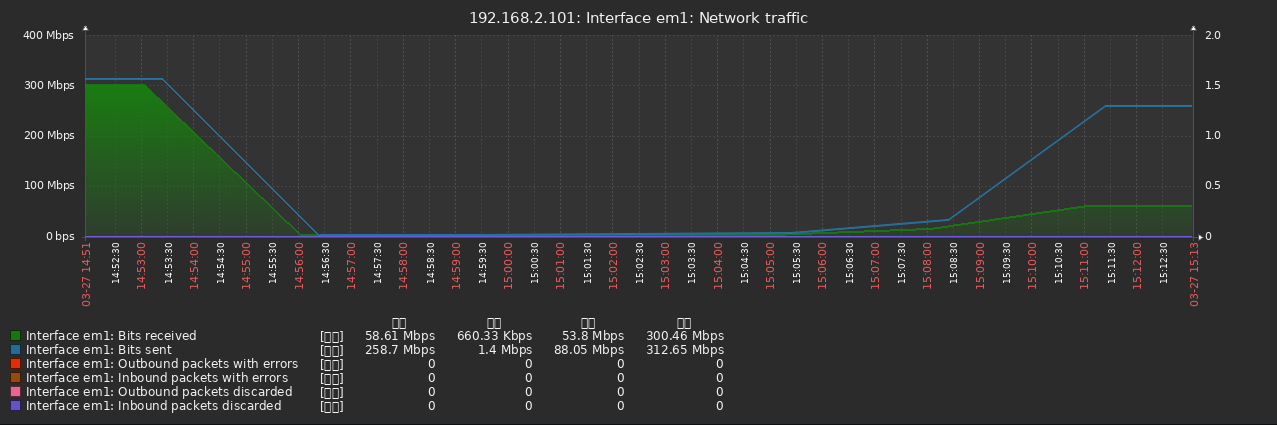
1. 在不进行预分区时，吞吐量和延迟表现极差；
2. 在小数据量下，预分区数量比较多的情况下，读内存和读磁盘性能差异不明显；



图表 3：不预分区时，节点的磁盘利用率



图表 4：不预分区时，节点的cpu使用率



图表 5：不预分区时，节点的网络流量情况

从上面三张图可以看出：

1. 在不进行预分区时，数据从磁盘读取，对磁盘造成了很大的压力，同时，大量的cpu时间都在等待io，这是吞吐量低、延迟高的主要原因；
2. **插入**

图表 6

从上图可以得出：

1. 在不进行预分区时，吞吐量是有预分区的表的一半；
2. 在不进行预分区时，延迟是有预分区的表的一倍；
3. 预分区数量比较多时，延迟反而比较低；

#### 小结

1. 不进行预分区时，无论是读性能还是写性能都比较差；
2. 在预分区数量比较多时，读磁盘的吞吐量比较高，写磁盘的延迟比较低；

### 不同压缩算法的性能对比

#### 方案

|  |  |
| --- | --- |
| 配置项 | 说明 |
| 数据条数 | 2000万 |
| 读/写次数 | 每客户端各执行666万6667次 |
| 预分区数量 | 15个 |
| 列族 | 1个 |
| 列族压缩算法 | 自变量 |
| rowkey大小 | 14字节 |
| 字段数量 | 10个 |
| 字段大小 | 10字节 |
| 请求分布模型 | zipfian |
| 客户端数量 | 3个 |
| 线程数 | 50 |
| 统计指标 | 吞吐量取三者平均 |
| 延迟取三者平均 |

#### 测试结果

**一、数据量**

图表 7

从上图可以看出：

1. 使用压缩后，数据量大幅降低；
2. 其中，gz算法压缩率最高，snappy次之，lz4压缩率最低；

**二、读磁盘**

图表 8

从上图可以看出：

1. 不使用压缩的读性能反而远远不如使用压缩；
2. 在小数据量规模下，这几种压缩算法的读取性能差异并不明显；
3. **插入**

图表 9

从上图可以看出：

1. 不使用压缩的写入性能依然不如使用压缩；
2. 在小数据量规模下，这几种压缩算法的写入性能差异并不明显；

#### 小结

1. 使用压缩后，读性能之所以比不使用压缩高，是因为：
   1. 压缩后，每次能读的数据比较多，变相提高BlockCache缓存命中率；
2. 使用压缩后，写性能之所以比不使用压缩高，是因为：
   1. 不使用压缩，写入的数据比较多，导致HBase频繁进行minor compaction，因此，延迟更高、吞吐量降低；

### 三、不同编码算法的性能对比

#### 方案

|  |  |
| --- | --- |
| 配置项 | 说明 |
| 数据条数 | 2000万 |
| 读/写次数 | 每客户端各执行666万6667次 |
| 预分区数量 | 15个 |
| 列族 | 1个 |
| 列族压缩算法 | snappy |
| 列族编码算法 | 自变量 |
| rowkey大小 | 14字节 |
| 字段数量 | 10个 |
| 字段大小 | 10字节 |
| 请求分布模型 | zipfian |
| 客户端数量 | 3个 |
| 线程数 | 50 |
| 统计指标 | 吞吐量取三者平均 |
| 延迟取三者平均 |

#### 测试结果

1. **数据量**

图表 10

从上图可以看出：

1. 开启数据编码（DATA\_BLOCK\_ENCODING）之后，数据大小比起不开数据编码有所下降；
2. 在此测试场景中，两种编码算法压缩比不明显；
3. **读磁盘**

图表 11

从上图可以看出：

1. 开启数据块编码后，读取性能比起不开数据块编码有不小的提升；
2. 在此测试场景，FAST DIFF算法的表现要略好于DIFF算法；
3. **插入**

图表 12

从上图可以看出：

1. 开启数据块编码后，插入性能比起不开数据块编码有一定的提升；
2. 在此测试场景中，不管开不开启数据块编码、使用哪种编码算法，性能差距都不大；

#### 小结

1. 开启数据块编码后，数据大小有一定的下降；
2. 开启数据块编码后，数据读取性能有不小的提升；

### 不同并发下的读写性能对比

#### 方案

|  |  |
| --- | --- |
| 配置项 | 说明 |
| 数据条数 | 10亿 |
| 读/写次数 | 每客户端读66万6667次、写666万6667次 |
| 预分区数量 | 128个 |
| 列族 | 1个 |
| 列族压缩算法 | snappy |
| 列族编码算法 | FAST\_DIFF |
| rowkey大小 | 14字节 |
| 字段数量 | 10个 |
| 字段大小 | 10字节 |
| 请求分布模型 | zipfian |
| 客户端数量 | 3个 |
| 线程数 | 自变量 |
| 总数据量 | 253.7GB |
| 统计指标 | 吞吐量取三者平均 |
| 延迟取三者平均 |

#### 测试结果

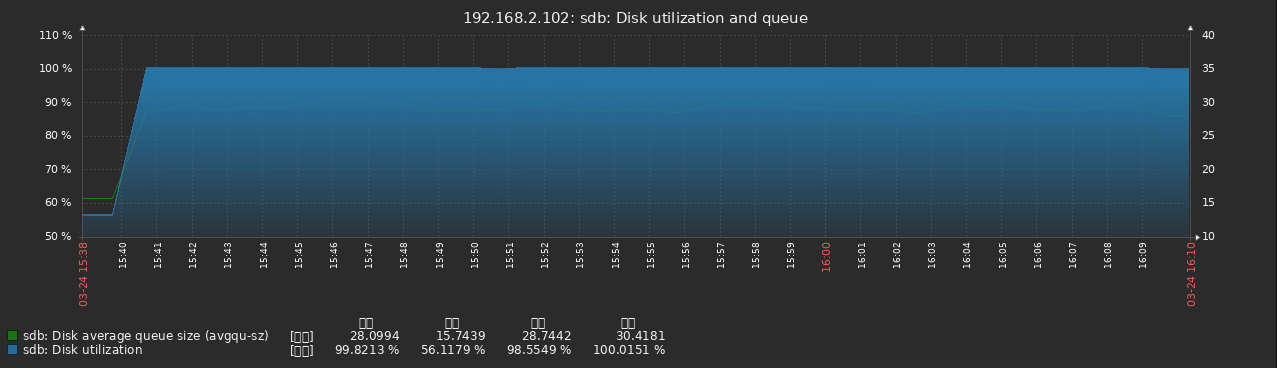
1. **读取**

图表 13

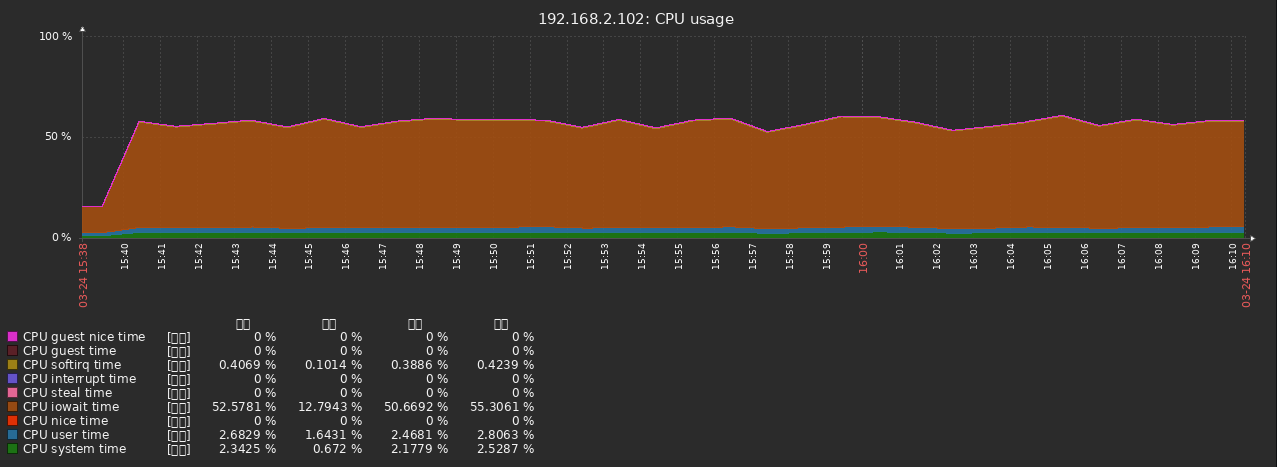
图表 14

从上面两张图可以看出：

1. 在没有缓存预热的情况下，大数据量下的查询操作非常慢，提高并发数并没有很好的提升吞吐量，反而大大提高了延迟；
2. 在进行一定程度的缓存预热后，吞吐量提升明显，但同样的，随着并发数的提高，延迟随着提高；



图表 15：无缓存情况下，并发数为10的磁盘利用率



图表 16：无缓存情况下，并发数为10的cpu使用率

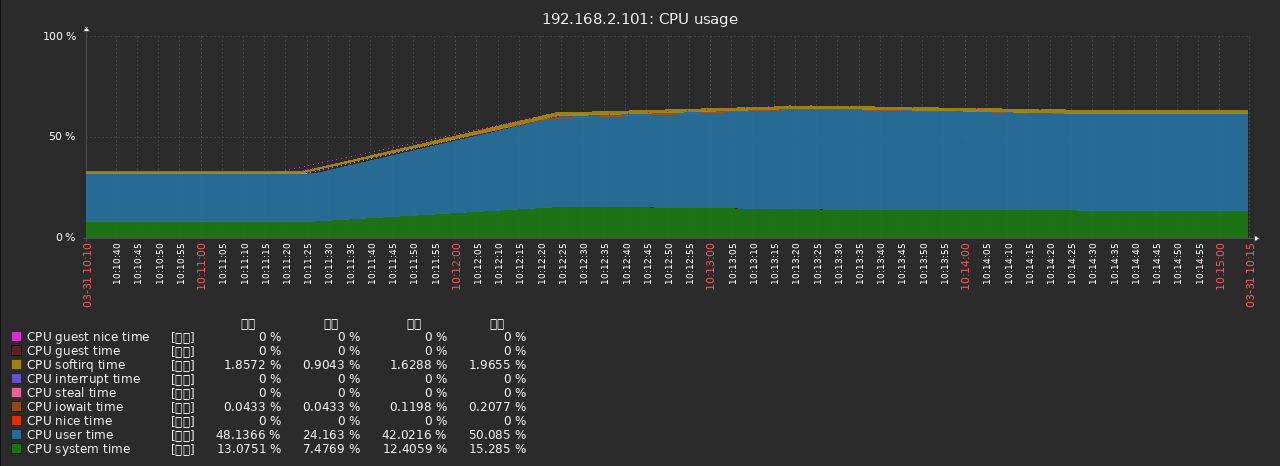
从上面两张图可以看出：

1. 磁盘利用率达到100%，证明HBase正大量从磁盘读数据；
2. 并发数并不高，因此cpu使用率也不高，大部分cpu时间都是在等待磁盘io；
3. **插入**

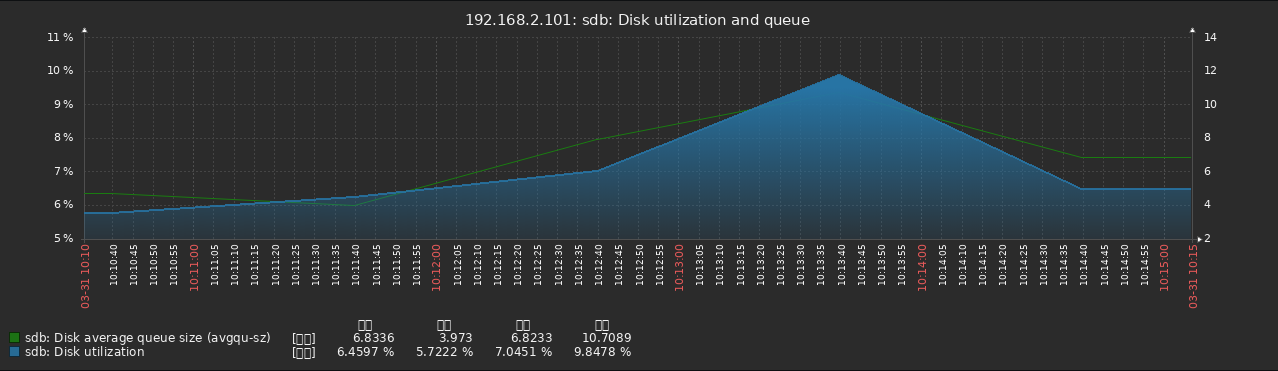
图表 17

从上图可以看出：

1. 随着并发数的提高，吞吐量随着提高，但到并发数为100之后，吞吐量提升不明显；
2. 随着并发数的提高，延迟随着变高，且延迟变高很明显；



图表 18：并发数为500时，cpu使用率



图表 19：并发数为500时，磁盘利用率

从上面两张图可以看出：

1. cpu大部分时间都是user time，很少出现io wait和软中断；
2. 磁盘利用率之所以突然飙高，是因为HBase把内存中的数据刷写到磁盘；

#### 小结

1. 对于读操作，性能瓶颈在于多少请求最终去读磁盘，对于非完全内存读操作来说，并发数带来的提升不明显；
2. 对于写操作，并发数带来的提升比较明显，但也有一定的上限；

### 不同并发下的批量读写性能对比

#### 方案

|  |  |
| --- | --- |
| 配置项 | 说明 |
| 数据条数 | 10亿 |
| 读/写次数 | 每客户端读66万6667次、写666万6667次 |
| 预分区数量 | 128个 |
| 列族 | 1个 |
| 列族压缩算法 | snappy |
| 列族编码算法 | FAST\_DIFF |
| rowkey大小 | 14字节 |
| 字段数量 | 10个 |
| 字段大小 | 10字节 |
| 请求分布模型 | zipfian |
| 客户端数量 | 3个 |
| 线程数 | 自变量 |
| 总数据量 | 253.7GB |
| 统计指标 | 吞吐量取三者平均 |
| 延迟取三者平均 |

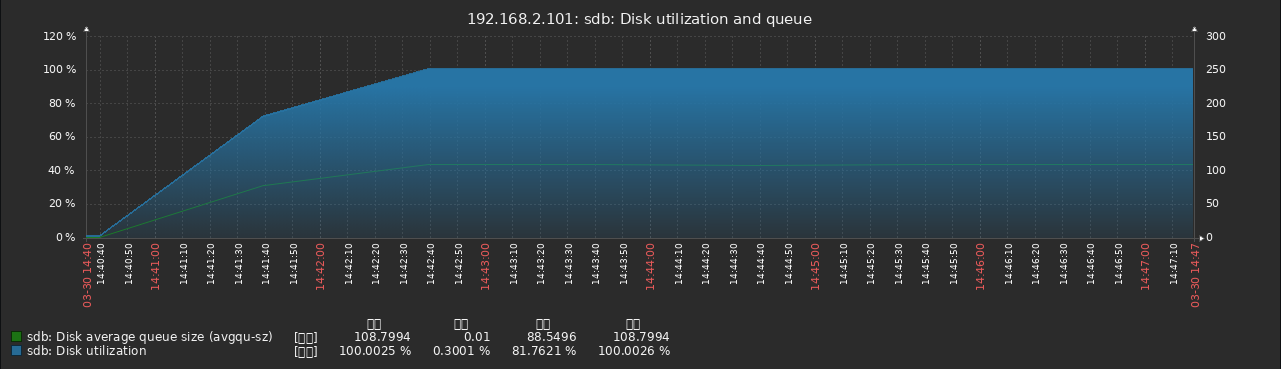
#### 测试结果

1. **批量读取**

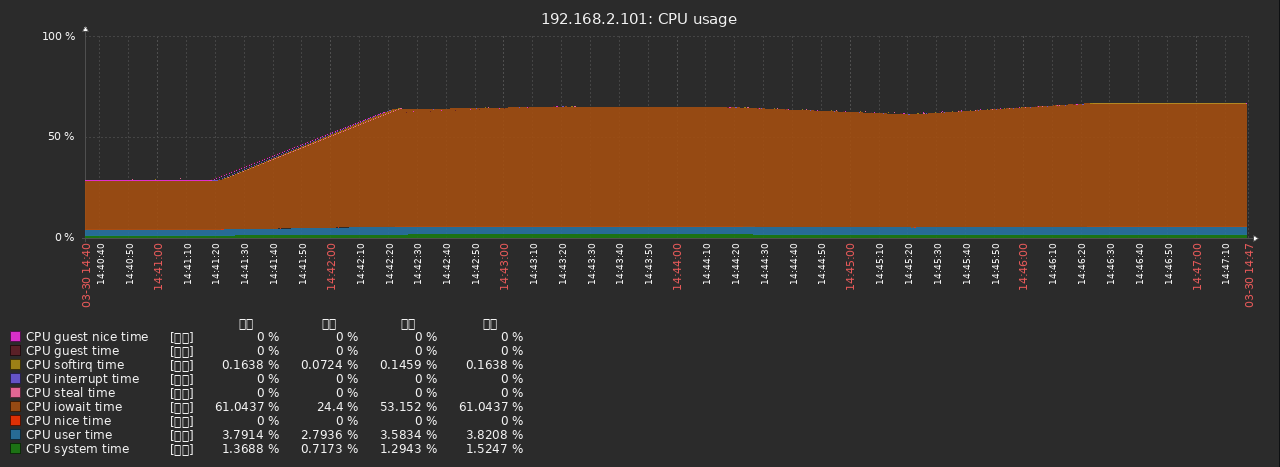
图表 20

从上图可以看出：

1. 在并发数为20时，bulkread操作的吞吐量达到最高，并发数为20~100时，吞吐量差距并不大，且整体不如有缓存预热下的read操作；
2. 随着并发数的增多，延迟随着变高，且延迟非常大，在不可接受的范围内；



图表 21：并发数为20时，磁盘利用率



图表 22：并发数为20时，cpu使用率

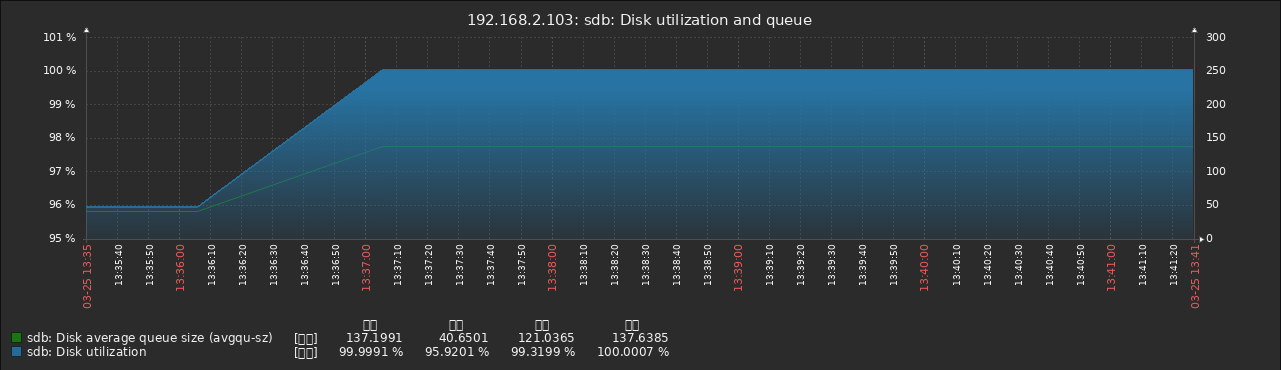
从上面两张图可以看出：

1. 在大数据量前提下，大部分的read操作都是读磁盘，批量读操作节省的rpc通信成本远远低于等待读磁盘带来的开销，性能反而不如单条数据的read操作；
2. 由于cpu速度比磁盘快，并发数越高，就有越多的线程在等待磁盘io，导致延迟变高；
3. **范围读取**

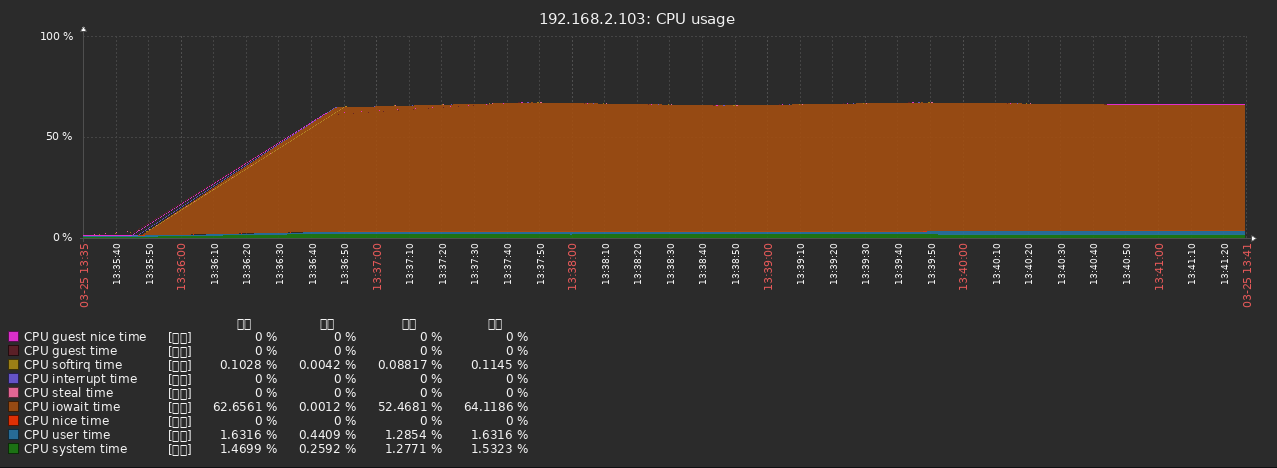
图表 23

从上图可以看出：

1. 在并发量20以上之后，整体吞吐量维持在一个相近的范围内；
2. 随着并发数的提升，延迟随着提高；



图表 24：并发数为50时，磁盘利用率



图表 25：并发数为50时，cpu使用率

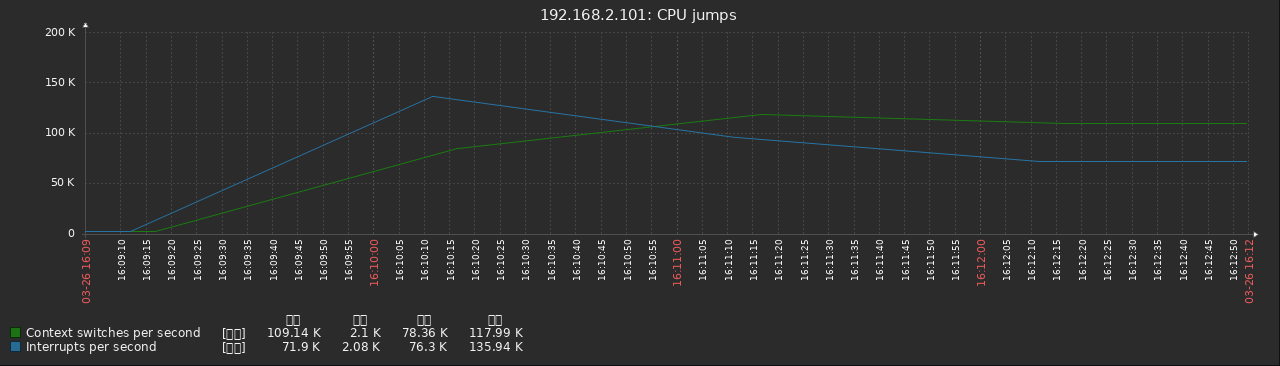
从上图可以看出：

1. Scan操作瓶颈依然在磁盘IO上；
2. 并发数越高，进行IO wait的线程就越多，不但不会提高吞吐量，还会提升延迟；
3. **批量插入**

图表 26

从上图可以看出：

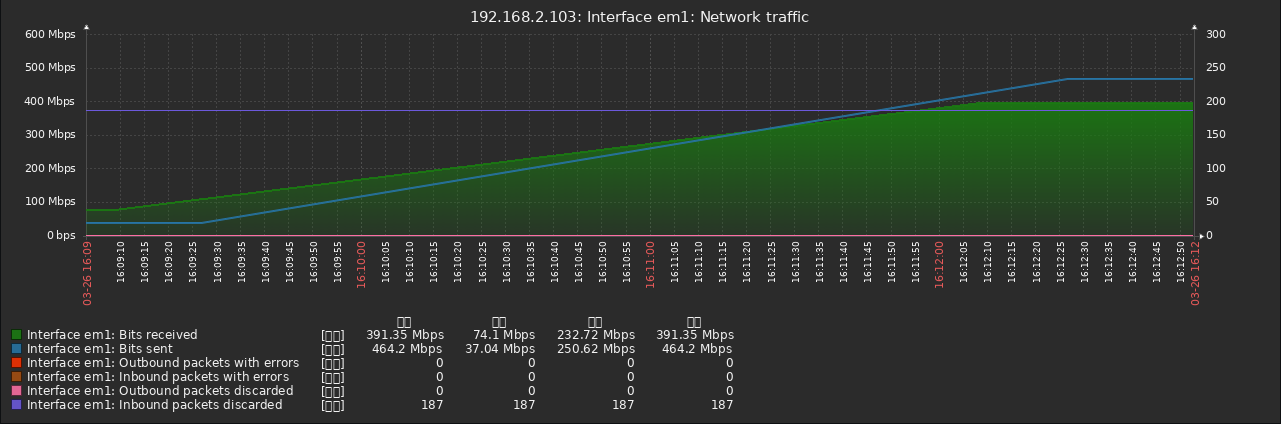
1. 批量插入对于吞吐量的提升非常明显，在当前测试环境下，每秒钟基本能插入30万条记录左右；
2. 批量插入导致了延迟大幅的提高，并且随着并发数的提高，延迟也随着提高；



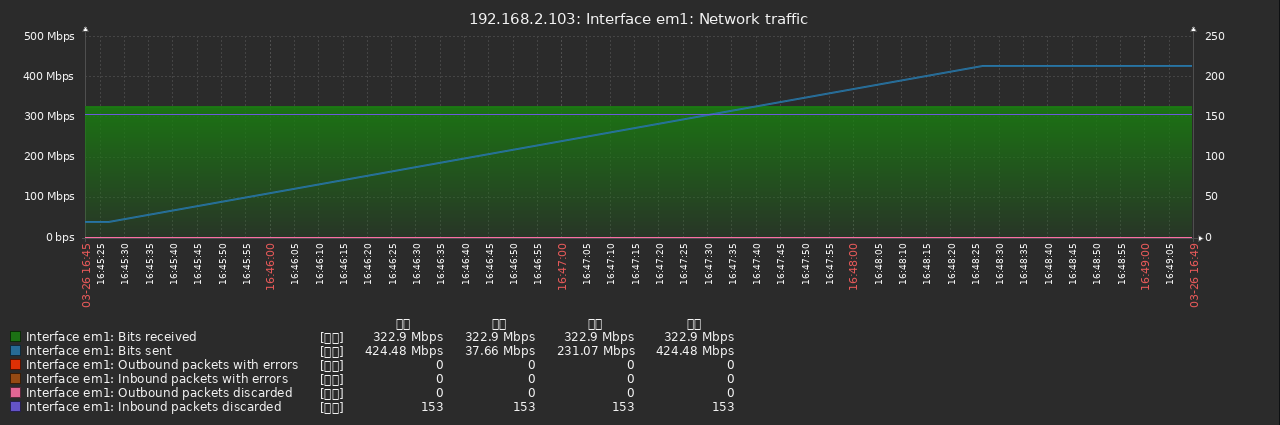
图表 27：并发数为500时，cpu jumps

从上图可以看出：

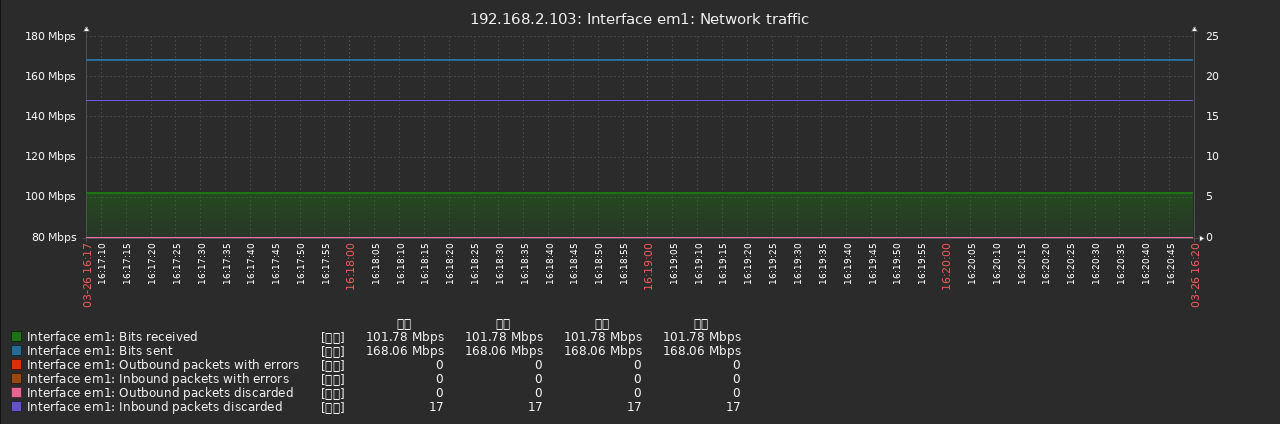
1. 在并发数为500的情况下，cpu上下文切换频繁；
2. 在并发数为500的情况下，cpu中断频繁；



图表 28：并发数为500时，网卡流量情况



图表 29：并发数为200时，网卡流量情况



图表 30：并发数为100时，网卡流量情况

从上图可以看出：

1. 并发数越高，可能导致网络越繁忙，在网络繁忙的情况下，可能出现丢包，会影响到整体的性能；
2. 网络流量远远还没达到网卡的上限；

#### 小结

1. 对于批量读操作，要分开来看：
   1. 对于小表，大部分数据能被内存容下，只有少部分数据会读磁盘的话，可以使用批量读操作；
   2. 对于大表，大部分数据都会读磁盘，此时使用批量读操作反而得不偿失；
2. 对于scan操作，哪怕是读磁盘，性能表现也很好，在合适的场景，推荐使用；
3. 对于批量写操作，性能整体表现也要好于单条写操作，推荐使用，需要注意的是，并发数没必要设置太高；

### 不同并发下的混合读写性能对比

#### 方案

|  |  |
| --- | --- |
| 配置项 | 说明 |
| 数据条数 | 10亿 |
| 读/写次数 | 每客户端各执行66万6667次 |
| 预分区数量 | 128个 |
| 列族 | 1个 |
| 列族压缩算法 | snappy |
| 列族编码算法 | FAST\_DIFF |
| rowkey大小 | 14字节 |
| 字段数量 | 10个 |
| 字段大小 | 10字节 |
| 请求分布模型 | zipfian |
| 客户端数量 | 3个 |
| 线程数 | 自变量 |
| 总数据量 | 253.7GB |
| 是否缓存预热 | 是 |
| 统计指标 | 吞吐量取三者平均 |
| 延迟取三者平均 |

#### 测试结果

1. **读写平衡**

图表 31

从上图可以看出：

1. 随着并发数的提高，吞吐量有所提高，但整体的吞吐量并不高，主要瓶颈还是在读操作上；
2. 基本上无论是插入还是查询，延迟都随着并发数的提高而提高；
3. **查询为主**

图表 32

从上图可以看出：

1. 随着并发数的提高，吞吐量不断提高，并且提升幅度较为客观；
2. 随着并发数的提高，延迟也不断提高，整体延迟都在100毫秒以内；
3. **插入为主**

图表 33

从上图可以看出：

1. 随着并发数的提高，吞吐量随着提高，但整体吞吐量并不高；
2. 随着并发数的提高，延迟随着提高，在并发数比较的情况下，延迟甚至接近一秒；

#### 小结

1. 本次测试印证了“HBase插入性能强，查询性能弱”的特点，特别是在数据量比较大的场景下，这个问题更为突出；
2. HBase更适合于大数据量写多读少的场景；

## 总结

根据以上测试结果和资源利用情况可以得出如下几点：

1. 选择一个合适的预分区数量，对读写性能有比较的提升；

不进行预分区的话，会出现热点问题，热点问题可能会造成性能瓶颈。因此，需要进行预分区，但是，HBase的机制决定了预分区越多，内存开支越大。在存在其他表读写的情况下，预分区越多，可能导致的内存交换越频繁，反而可能得不偿失。

1. 使用压缩对存储空间节约较大，对读写也有一定程度的提升；

开启数据压缩后，能够解决大量的存储空间。虽然在读写数据过程中耗费了更多的cpu时间，但在写入的时候，由于数据量少了，发生minor compaction次数变少，在查询过程中，一次性读取的数据量变多，变相提高了缓存命中率。所以，总体来看，使用压缩是比较划算的。

1. 开启数据块编码后，能够节约一定的存储空间，对读写有一定的提升；
2. 并发数的提高，能一定程度提高读写性能；

并发数的提高，能够一定程度提升读写性能。但对于大数据量的查询来说，并发数提高带来的提升不明显。对于插入操作，性能表现比较稳定，在大部分情况下，都能维持1万以上的吞吐量。随着并发数的提高，吞吐量有所上升。并发数提高带来的一个后果就是延迟会变高；

1. 总的来看，HBase更适合写多读少的场景；

由于HBase写性能好于读性能，因此更适合写多读少的场景；

**后续工作**

本次测试仅仅完成了HBase基本的读写性能的测试，实际上，HBase还提供诸如异步读写、中等大小对象MOB（Storing Medium-sized Objects）存储等特性。后续工作将对这些特性补充测试。