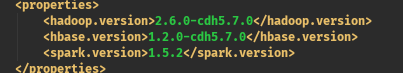
HBase get性能测试报告

服务器配置如下： 内存256G CPU：Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz

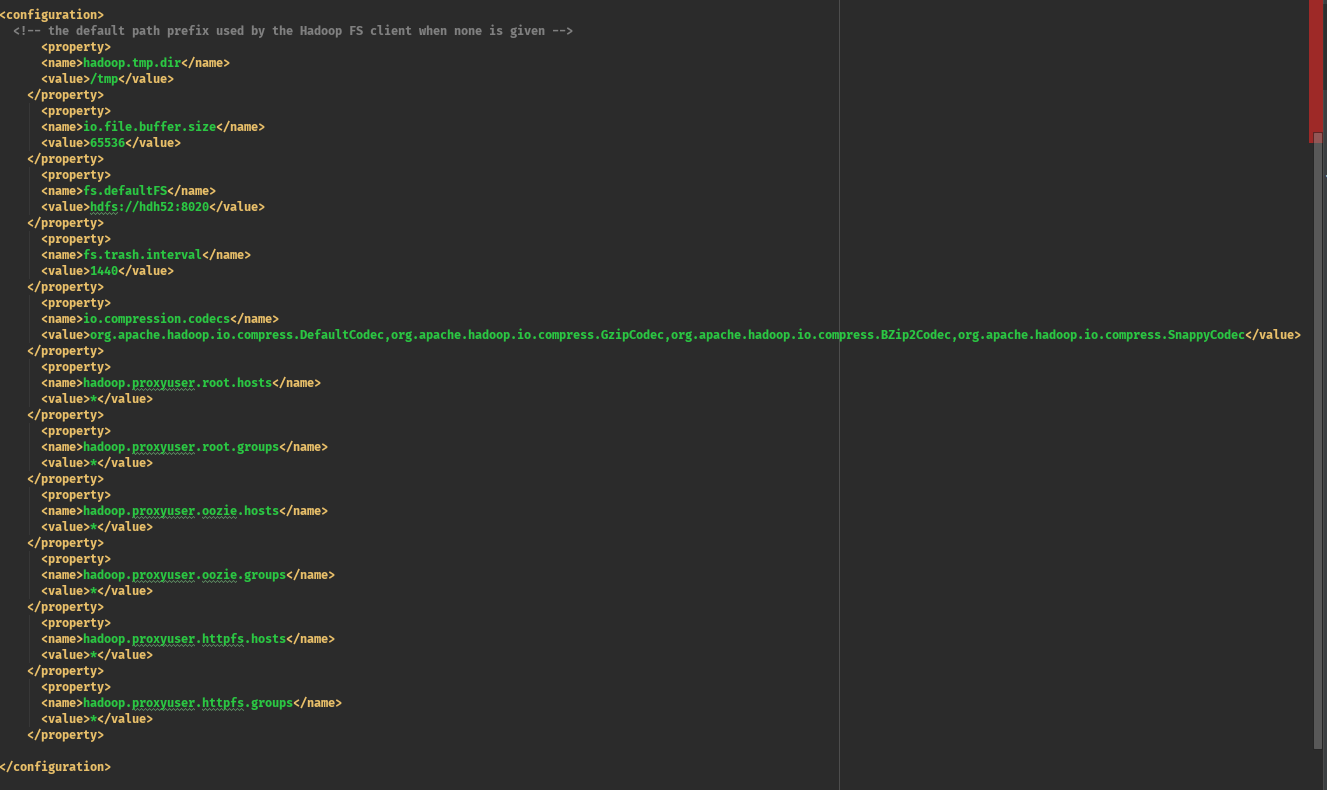
CPU个数： 2个 CPU cores：10

基于如下组件版本：

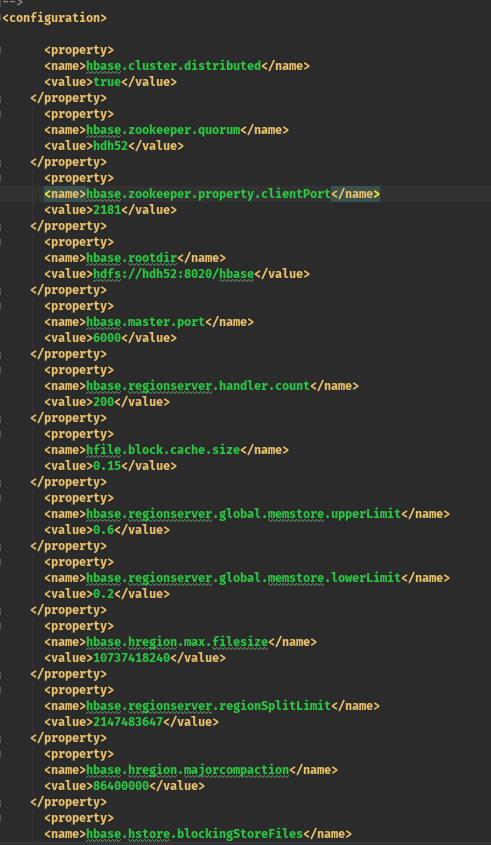


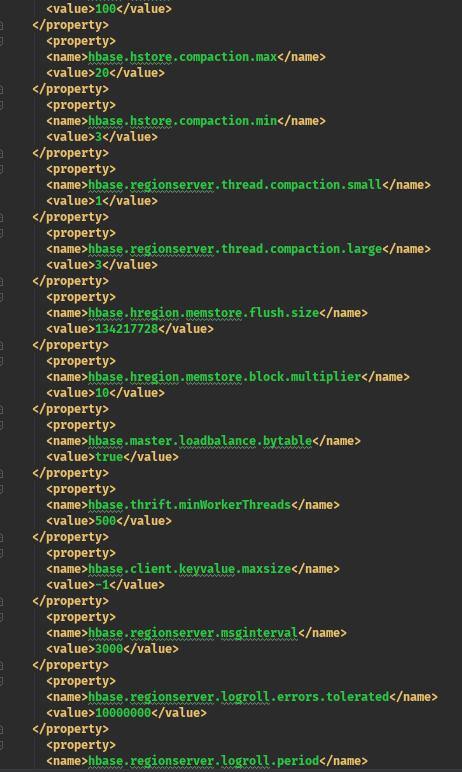
原来的配置清单如下：

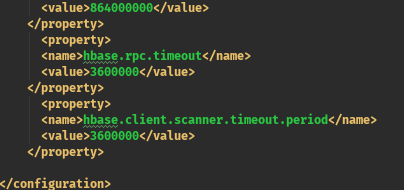
Hadoop core-site.xml 文件配置如下



hbase-site.xml 配置如下：







思路：

1. 理清当前大数据环境和HBase相关的配置
2. 在当前配置下用HBase的client，编写java代码，Get函数去多次获取HBase中的一列，记录平均耗时。Scan函数固定起始列和终止列多次执行计算平均耗时。
3. 结合HBase的原理，控制变量，修改配置后重复2步骤，记录。

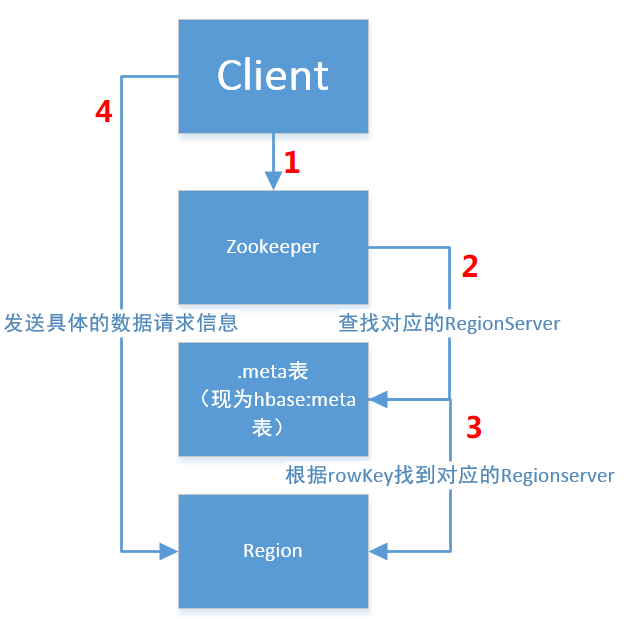
注意要点，此次目的在于优化实际项目的HBase Get耗时。所以一切尽量以实际项目的环境来。但是

1.由于资源紧张，只有一台服务器。也就意味着是本次测试是在伪分布模式下进行的。

2.现场的数据量由于不同项目会有差别，在测试的时候，数据只能将尽可能的多。因此会与实际项目有所不同。这样会影响到Region的数量分布。

基本理论：

HBase的读取操作，流程大致如下：



n表中KeyValue条目数量（包括Put的结果和Delete留下的墓碑标记）

b HFile里数据块（HFile Block）的数量

e 平均一个HFile里KeyValue条目的数量（如果你知道行的大小，可以计算得到）

c 每行里列的平均数量

1. 客户端寻找正确的RegionServer和Region花费3次固定运算找到正确的Region

查找ZK，查找meta-region-server 查找HBase:meta 是一次O（1）运算

1. ·在指定的Region上，行在度过程中可能存在于两个地方：如果还没有刷写到磁盘中就位于MenStore，如果已经刷写则位于HFile里。
2. 我们用e合理代表任何指定时间在MemStore里的条目数量。如果一行在MemStore里，因为MemStore里使用的是跳表（skip list）实现的，所以查找的时间复杂度是O（loge）。

如果一行已经被刷写到磁盘上，你需要找到正确的HFile数据块。数据块索引是排过序的，所以查找正确的数据块是一次时间复杂度为O（Log b）的运算。查找行里的KeyValue对象是在数据块里的一次线性扫描操作。在你找到的第一个KeyValue对象后，随后查找剩下的对象就是一次线性扫描操作。假设行里的单位都在一个数据块里。扫描的时间复杂度就是O(el b). 如果行里的单元不在同一个数据块里，这种扫描需要访问多个数据块里的数据，所以这时候的运算由读取的行数决定，其时间复杂度为O（c）。也就是说这会在哪个扫描的时间复杂度为O（max （c，el b））

查找的耗时公式：

**O（1） + 【O （log e）或 O （1）】+ O（log b） + O （max （c， el b））**

因为HBase：Meta表会在第一次读取的时候缓存进入客户端，因此实际影响的时间主要为**【O （log e）或 O （1）】+ O（log b） + O （max （c， el b））。**决定性因素为扫描HFile数据块找到相关KeyValue对象所花费的时间。

因为在查找数据时，首先是从MemStore里找，找不到后再去StoreFile上读。

简单的认为可以将MemStore的空间设置尽可能的大。

将这个读取流程更加细化。

## 一、客户端优化

1.读取请求从客户端出发，这里客户端有个*hbase.client.scanner.caching*

值，即scan的缓存设置。因为照顾到一次性读取可能会将带宽严重消耗，影响其他的通信，或者是导致客户端发生OOM。因此读取一般是通过RPC请求多次返回的。默认设置为100.

如果每次请求总是大于这个值，可以考虑将这个值调大。

对于Get请求，也是可以考虑通过批量的方式来获取。 Get[table.get(List<Get>)]

以上都是为了减少RPC的次数，来加快读取数据的速度。

2.在读取时客户端还有一个参数能够设置 blockCache。就是指读取出来的数据是否要存入缓存之中。这个得根据业务需求来判断，不可一概而论。因为如果是经常性读取的 数据放入缓存之后自然很好。能大大加快读取速度。但是不是那么读取频繁的数据大量加入缓存之中的话，会将原本的热度数据给挤出缓存，反而会使体验变差。

接下来请求进入服务器端。

## 二、服务器端优化

1.服务器端主要是对集群的效率进行优化。

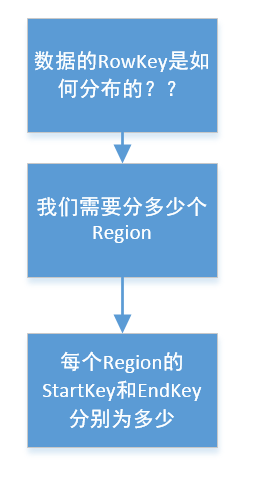
例如，读取请求是否都集中在某一两个RegionServer上，这个和数据倾斜一样，无法发挥出集群原本的性能。

这里可以通过查看所有RegionServer的读取QPS曲线来做判断。监听QPS曲线的工具推荐为Jmeter.附录中会教你如何使用Jmeter。

如果这一步看到确实存在这种读取请求集中在某些RegionServer的现象时，建议是通过对RowKey进行一些散列化操作，来使得尽可能的分布到不同的机器上去。

并且将Region进行预分区处理。

Regin预分区主要是以下思路



2.配置BlockCache占比与策略

BlockCache即读取时的缓存，默认情况下是和写入时的MemStore配置是一样的都为40%。

根据现实情况，读写业务的多少来调整配置。过车记录按理来说应该是写入比较多，因此咋内存有限的情况下，还是应该将较多的内存分配给MemSotre。另外能够调整的还有BlockCache的策略模式。不同的策略对于读取性能其实没有太多的影响，主要影响的是GC。

根据网上的资料，如果缓存的命中率低，则可以选择offheap模式，当JVM内存配置量< 20G时，可以选择LRUBlockCache模式。根据我们的服务器来看，应当换成offheap模式。

3.来请求回到了HFile中寻找符合条件的数据项。此时HFile的数量也直接影响到了读取速度。这想必很好理解。HFile的文件越多，这IO时间越长。因此有必要调整Compaction的执行策略。

主要是两个参数。

Hbase.hstore.compactionThreshold和hbase.hstore.compaction.max.size 前一个指一个Store中文件数超过多少进行合并操作。后一个指参与合并的文件大小最大不能超过多少。

1. hbase-site中的设置

这个文件中主要有hbase.regionserver.global.memstore.szie和hfile.block.cache.size两个参数前者表示regionserver中有多少内存分配给写缓存，那么剩下来的就是读缓存。后者表示hfile中有多少内存分配给读缓存。

## 三、HBase族列设计

HBase的族列也会影响到读取的效率。这里有必要开始BloomFilter过滤器。BloomFilter过滤器也有一项需要设置。即过滤的

关于HBase列族的优化主要就是开启BloomFilter。将BloomFilter设置为ROW。查看了现在的HBase表，发现已经设置了此项。

## 四、在HDFS端优化

也能进行优化。HBase的数据最终还是保存在HDFS系统里面，最后肯定是要与HDFS进行交互的。

开启Short-CircuitLocal Read功能：

一般而言，HDFS读取数据的流程是这样的。

客户端向DataNode发送读取数据的请求，DataNode接收到请求之后从硬盘里将数据读取出来通过TPC发送给客户端。然而开启了Short-CircuitLocal Read功能之后可以绕过DataNode环节直接去本地数据中读取。当然本地不一定能有需要读取的数据，这就涉及到了下一个设置项。

在hdfs.site-xml

添加

<property>

<name>dfs.client.read.shortcircuit</name>

<value>true</value>

</property>

开启HedgedRead功能：

HBase在HDFS中保存的数据一般而言都有三份。优先通过Short-CircuitLocal Read功能读取本地数据，如果本地不存在则通过HedgedRead功能去读取

向多个dataNode发送数据请求，哪个先返回则删除另外的2个。

这边就有部分时间损耗在网络请求和IO损耗上了。

# 实验部分

之前提到过这是一个单机环境。那么有些优化项可能无法进行。有些就算验证了也不具备说服力。在此先做一个说明。

首先查看客户端的设置。因为是通过Java API去获取数据，所以首先去看下Java的调用函数。

如果是Get操作。HTable的get函数用两种，一种是直接传入单个Get对象，即只获取一条数据。另一种是传入List<Get>，即传入多个Get对象。其主要区别在于批处理读取只需要一次网络I/O开销。所以如果需要读取多条数据的时候尽可能的使用get(List<Get>)函数。

另外Get函数尽量在设置的时候将需要的family和 Column设置进去，这样在搜索以即返回的时候能够减少返回的数据量。

Scan的话主要设置*hbase.client.scanner.caching*这个属性。在hbase-site.xml文件中设置。

根据了解，目前项目中比较慢的是Get请求。在用户根据一定条件检索过车数据时。先是通过ES筛选出了符合条件的rowkey再根据rowkey去HBase中获取相应的数据记录。

那么基本上rowkey是不会连续的，所以必然要用到htable.get(List<Get>)方法。

接着我现在当前配置下也用YCSB 先做了一次性能测试。

（YSCB是Yahoo开源的一款测试云服务的工具。目前支持多款数据库，操作也很简单）。

首先执行

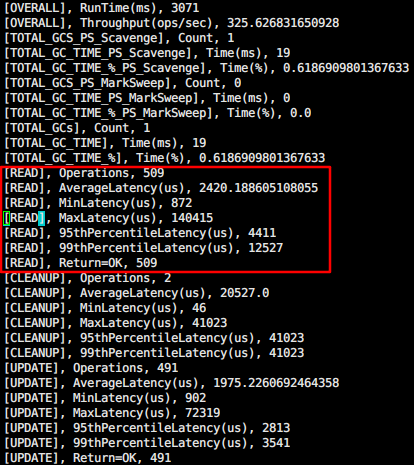
bin/ycsb load hbase12 -P workloads/workloada -p threads=10 -p table=BAYONET\_VEHICLEPASS -p columnfamily=cf -p recordcount= 8775543 -s > load.log

将数据加载入HBase 数据量为8775543。

执行bin/ycsb run hbase12 -P workloads/workloada -p table=BAYONET\_VEHICLEPASS -p columnfamily=cf -s > run.log

混合了50%的读和50%的写

测试结果如下：



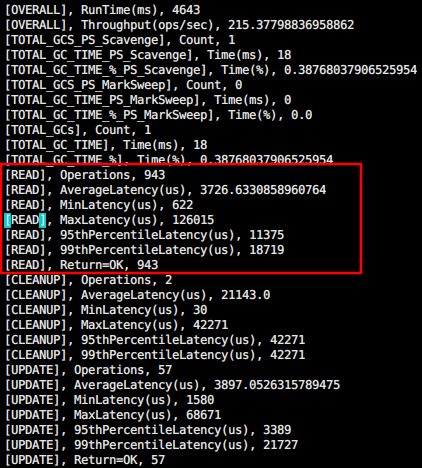
数值为时间 单位为微秒 1秒 = 100万微秒

所以大致的平均读取速度为0.002秒。 最长耗时为0.14秒

执行 bin/ycsb run hbase10 -P workloads/workloadb -p table=BAYONET\_VEHICLEPASS -p columnfamily=cf -s > run.log

混合了95%的读和5%的写，该workload侧重于测试集群的读能力

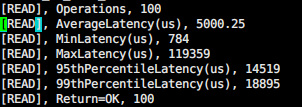
测试结果如下：

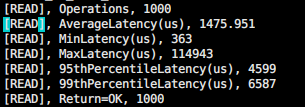


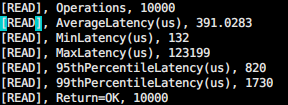
可以看到读取的平均耗时约为0.0037秒。 最长耗时为0.12秒。在误差范围之内。

我尝试在HBase-site.xml 修改file.block.cache.size 和 hbase.regionserver.global.memsotre.size 属性，分别设置为0.79和0.01.。因为这两者的总和不能超过80%。故按此设置。其中file.block.cache.size是指

测试结果如下：







分别执行读取100条， 1000条，10000条数据的操作可以看到

整个平均读取耗时是明显下降了的。

与之前没有修改是相比，读取耗时也是有所降低。

对于这种现象可以假设，读取时有一段固定的耗时，当每增加一条读取数据的时候，实际增加的边际效益是不高的，那么当读取条数变多时，每一条读取的耗时反而是下降了。

但是这种情况对于实际的项目中貌似无用，因为项目中看的是总耗时。看上面的图片，当读取条数增加的时候，总耗时也是在增加的。