Hadoop 集群（第 12 期副刊）

——HBase 性能优化

1、从配置角度优化

1.1 修改Linux配置

Linux 系统最大可打开文件数一般默认的参数值是 1024，如果你不进行修改并发量上来的时候会出现“Too Many Open Files”的错误，导致整个 HBase 不可运行，你可以用 ulimit -n 命令进行修改，或者修改/etc/security/limits.conf 和/proc/sys/fs/file-max 的参数，具体如何修改可以去 Google 关键字 “linux limits.conf ”

1.2 修改 JVM 配置

修改 **hbase-env.sh** 文件中的配置参数，根据你的机器硬件和当前操作系统的 JVM（32/64 位）配置适当的参数：

HBASE_HEAPSIZE 4000　HBase 使用的 JVM 堆的大小
HBASE_OPTS　"﹣server　﹣XX:+UseConcMarkSweepGC"JVM GC 选项
HBASE_MANAGES_ZKfalse　是否使用 Zookeeper 进行分布式管理

1.3 修改HBase配置

- **zookeeper.session.timeout**

默认值：3 分钟（180000ms）

说明：RegionServer 与 Zookeeper 间的连接超时时间。当超时时间到后，ReigonServer 会被 Zookeeper 从 RS 集群清单中移除，HMaster 收到移除通知后，会对这台 server 负责的 regions 重新 balance，让其他存活的 RegionServer 接管.

调优：

这个 timeout 决定了 RegionServer 是否能够及时的 failover。设置成 1 分钟或更低，可以减少因等待超时而被延长的 failover 时间。

不过需要注意的是，对于一些 Online 应用，RegionServer 从宕机到恢复时间本身就很短的（网络闪断，crash 等故障，运维可快速介入），如果调低 timeout 时间，反而会得不偿失。因为当 ReigonServer 被正式从 RS 集群中移除时，HMaster 就开始做 balance 了（让其他 RS 根据故障机器记录的 WAL 日志进行恢复）。当故障的 RS 在人工介入恢复后，这个 balance 动作是毫无意义的，反而会使负载不均匀，给 RS 带来更多负担。特别是那些固定分配 regions 的场景。

- **hbase.regionserver.handler.count**

默认值：10

说明：RegionServer 的请求处理 IO 线程数。

调优：

这个参数的调优与内存息息相关。

较少的 IO 线程，适用于处理单次请求内存消耗较高的 Big PUT 场景（大容量单次 PUT 或设置了较大 cache 的 scan，均属于 Big PUT）或 ReigonServer 的内存比较紧张的场景。

较多的 IO 线程，适用于单次请求内存消耗低，TPS 要求非常高的场景。设置该值的时候，以监控内存为主要参考。

这里需要注意的是如果 server 的 region 数量很少，大量的请求都落在一个 region 上，因快速充满 memstore 触发 flush 导致的读写锁会影响全局 TPS，不是 IO 线程数越高越好。

压测时，开启Enabling RPC-level logging，可以同时监控每次请求的内存消耗和GC的状况，最后通过多次压测结果来合理调节IO线程数。

这里是一个案例：Hadoop and HBase Optimization for Read Intensive Search Applications，作者在SSD的机器上设置IO线程数为 100，仅供参考。

- **hbase.hregion.max.filesize**

默认值：256M

说明：在当前 ReigonServer 上单个 Reigon 的最大存储空间，单个 Region 超过该值时，这个 Region 会被自动 split 成更小的 region。

调优：

小 region 对 split 和 compaction 友好，因为拆分 region 或 compact 小 region 里的 storefile 速度很快，内存占用低。缺点是 split 和 compaction 会很频繁。

特别是数量较多的小 region 不停地 split, compaction，会导致集群响应时间波动很大，region 数量太多不仅给管理上带来麻烦，甚至会引发一些 Hbase 的 bug。

一般 512 以下的都算小 region。

大 region，则不太适合经常 split 和 compaction，因为做一次 compact 和 split 会产生较长时间的停顿，对应用的读写性能冲击非常大。此外，大 region 意味着较大的 storefile，compaction 时对内存也是一个挑战。

当然，大 region 也有其用武之地。如果你的应用场景中，某个时间点的访问量较低，那么在此时做 compact 和 split，既能顺利完成 split 和 compaction，又能保证绝大多数时间平稳的读写性能。

既然 split 和 compaction 如此影响性能，有没有办法去掉？

compaction 是无法避免的，split 倒是可以从自动调整为手动。

只要通过将这个参数值调大到某个很难达到的值，比如 100G，就可以间接禁用自动 split（RegionServer 不会对未到达 100G 的 region 做 split）。

再配合 RegionSplitter 这个工具，在需要 split 时，手动 split。

手动 split 在灵活性和稳定性上比起自动 split 要高很多，相反，管理成本增加不多，比较推荐 online 实时系统使用。

内存方面，小 region 在设置 memstore 的大小值上比较灵活，大 region 则过大过小都不行，过大会导致 flush 时 app 的 IO wait 增高，过小则因 store file 过多影响读性能。

- **hbase.regionserver.global.memstore.upperLimit/lowerLimit**

默认值：0.4/0.35

upperlimit 说明：hbase.hregion.memstore.flush.size 这个参数的作用是当单个 Region 内所有的 memstore 大小总和超过指定值时，flush 该 region 的所有 memstore。RegionServer 的 flush 是通过将请求添加一个队列，模拟生产消费模式来异步处理的。那这里就有一个问题，当队列来不及消费，产生大量积压请求时，可能会导致内存陡增，最坏的情况是触发 OOM。这个参数的作用是防止内存占用过大，当 ReigonServer 内所有 region 的 memstores 所占用内存总和达到 heap 的 40%时，HBase 会强制 block 所有的更新并 flush 这些 region 以释放所有 memstore 占用的内存。

lowerLimit 说明： 同 upperLimit，只不过 lowerLimit 在所有 region 的 memstores 所占用内存达到 Heap 的 35%时，不 flush 所有的 memstore。它会找一个 memstore 内存占用最大的 region，做个别 flush，此时写更新还是会被 block。lowerLimit 算是一个在所有 region 强制 flush 导致性能降低前的补救措施。在日志中，表现为 “** Flush thread woke up with memory above low water.”

调优：这是一个 Heap 内存保护参数，默认值已经能适用大多数场景。

参数调整会影响读写，如果写的压力大导致经常超过这个阀值，则调小读缓存 hfile.block.cache.size 增大该阀值，或者 Heap 余量较多时，不修改读缓存大小。

如果在高压情况下，也没超过这个阀值，那么建议你适当调小这个阀值再做压测，确保触发次数不要太多，然后还有较多 Heap 余量的时候，调大 hfile.block.cache.size 提高读性能。

还有一种可能性是 hbase.hregion.memstore.flush.size 保持不变，但 RS 维护了过多的 region，要知道 region 数量直接影响占用内存的大小。

- **hfile.block.cache.size**

默认值：0.2

说明：storefile 的读缓存占用 Heap 的大小百分比，0.2 表示 20%。该值直接影响数据读的性能。

调优：当然是越大越好，如果写比读少很多，开到 0.4-0.5 也没问题。如果读写较均衡，0.3 左右。如果写比读多，果断默认吧。设置这个值的时候，你同时要参考“hbase.regionserver.global.memstore.upperLimit”，该值是 memstore 占 heap 的最大百分比，两个参数一个影响读，一个影响写。如果两值加起来超过 80-90%，会有 OOM 的风险，谨慎设置。

- **hbase.hstore.blockingStoreFiles**

默认值：7

说明：在 flush 时，当一个 region 中的 Store（Coulmn Family）内有超过 7 个 storefile 时，则 block 所有的写请求进行 compaction，以减少 storefile 数量。

调优：block 写请求会严重影响当前 regionServer 的响应时间，但过多的 storefile 也会影响读性能。从实际应用来看，为了获取较平滑的响应时间，可将值设为无限大。如果能容忍

响应时间出现较大的波峰波谷，那么默认或根据自身场景调整即可。

- **hbase.hregion.memstore.block.multiplier**

默认值：2

说明：当一个 region 里的 memstore 占用内存大小超过 hbase.hregion.memstore.flush.size 两倍的大小时，block 该 region 的所有请求，进行 flush，释放内存。

虽然我们设置了 region 所占用的 memstores 总内存大小，比如 64M，但想象一下，在最后 63.9M 的时候，我 Put 了一个 200M 的数据，此时 memstore 的大小会瞬间暴涨到超过预期的 hbase.hregion.memstore.flush.size 的几倍。这个参数的作用是当 memstore 的大小增至超过 hbase.hregion.memstore.flush.size 2 倍时，block 所有请求，遏制风险进一步扩大。

调优： 这个参数的默认值还是比较靠谱的。如果你预估你的正常应用场景（不包括异常）不会出现突发写或写的量可控，那么保持默认值即可。如果正常情况下，你的写请求量就会经常暴长到正常的几倍，那么你应该调大这个倍数并调整其他参数值，比如 hfile.block.cache.size 和 hbase.regionserver.global.memstore.upperLimit/lowerLimit，以预留更多内存，防止 HBase server OOM。

- **hbase.hregion.memstore.mslab.enabled**

默认值：true

说明：减少因内存碎片导致的 Full GC，提高整体性能。

调优：

Arena Allocation，是一种 GC 优化技术，它可以有效地减少因内存碎片导致的 Full GC，从而提高系统的整体性能。本文介绍 Arena Allocation 的原理及其在 Hbase 中的应用 -MSLAB。

1）背景

假设有 1G 内存，我顺序创建了 1 百万个对象，每个对象大小 1K，Heap 会被渐渐充满且每个对象以创建顺序相邻。此时，如果我释放 50 万个奇数对象，即 1 3 5 7 后，剩余空间会多出 500M，而这段内存空间就不再连续了。问题出现？

如果我打算 new 一个 2K 大小的对象，JVM 将无从分配它，因为找不到连续可用的内存空间来容纳这个对象，就算 Heap 当时还有 500M 的剩余空间，也无能为力。最终，JVM 会选择触发 Full GC 重新压缩内存使之连续，然后再分配。

结论：触发 Full GC，并不只有在内存满或达到触发比例的时候，还有可能是因为内存碎片。

产生内存碎片的主要原因是：

- 分配的大小不一。
- 分配的空间不连续。

如何检测因内存碎片触发了 Full GC？

通过启动 java 时，添加 -XX:PrintFLSStatistics=1 参数来打印每次 gc 前后的 Heap 余量。较大的余量，可以怀疑 Heap 中存在内存碎片过多。

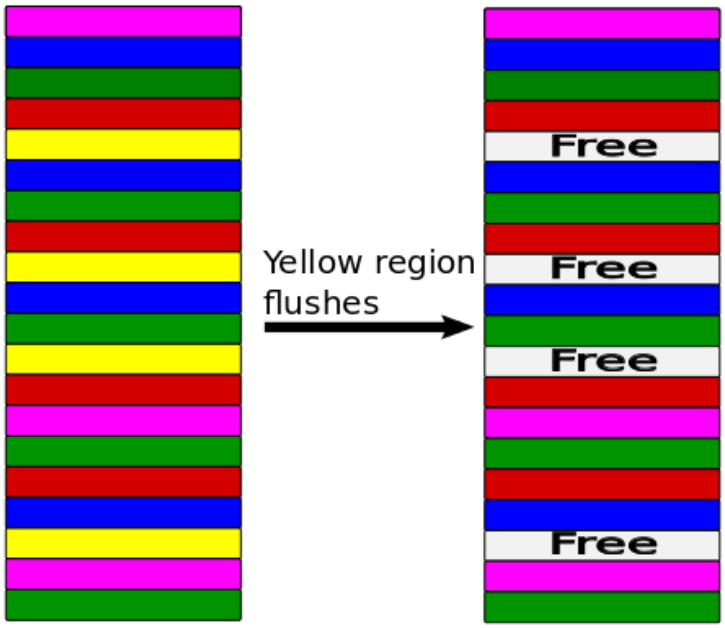
另外这篇 blog 有更详细的图文解释：

http://www.cloudera.com/blog/2011/02/avoiding-full-gcs-in-hbase-with-memstore-local-allocation-buffers-part-2/

2）HBase 中的内存碎片

HBase 为了提高写入性能，为每个 region 添加了一个内存写缓存-Memstore。当单个 Memstore 的大小达到 memstore.size 或 Heap 内存达到 hbase.regionserver.global.memstore.upperLimit/lowerLimit 百分比限制时，就会触发整个 region 的 flush，最终将所有数据写入 HDFS 并释放 region 下所有 Memstores 占用的内存（GC 不一定及时）。

Region flush 导致内存碎片的示意图：



左边五颜六色的是不同的 region 在内存中的位置，它是无序的，因为客户端的请求是无规律的。此时假设黄色的 region 触发了 flush，那么右边将会出现与之对应的多个空洞，即内存碎片。这张图以 **region** 为粒度，仅仅是为了更直观地表示这种现象。真实场景中，这些空洞是更细粒度的 KeyValue 级对象，它能直接导致创建对象时触发 Full GC。

3）Arena Allocation

Arena Allocation 是一种非传统的内存管理方法。它通过顺序化分配内存，内存数据分块等特性使内存碎片粗化，有效改善了内存碎片导致的 Full GC 问题。

它的**原理**：

- 创建一个大小固定的 bytes 数组和一个偏移量，默认值为 0。
- 分配对象时，将新对象的 data bytes 复制到数组中，数组的起始位置是偏移量，复制完成后为偏移量自增 data.length 的长度，这样做是防止下次复制数据时不会覆盖掉老数据（append）。
- 当一个数组被充满时，创建一个新的数组。
- 清理时，只需要释放掉这些数组，即可得到固定的大块连续内存。

在 Arena Allocation 方案中，数组的大小影响空间连续性，越大内存连续性越好，但内存平均利用率会降低。

4）HBase 的解决方案-MSLAB

MSLAB，全称是 MemStore-Local Allocation Buffer，是 Cloudera 在 HBase 0.90.1 时提交的一个 patch 里包含的特性。它基于 Arena Allocation 解决了 HBase 因 Region flush 导致的内存碎片问题。

MSLAB 的**实现原理**（对照 Arena Allocation，HBase 实现细节）：

- MemstoreLAB 为 Memstore 提供 Allocator。
- 创建一个 2M（默认）的 Chunk 数组和一个 chunk 偏移量，默认值为 0。
- 当 Memstore 有新的 KeyValue 被插入时，通过 KeyValue.getBuffer()取得 data bytes 数组。将 data 复制到 Chunk 数组起始位置为 chunk 偏移量处，并增加偏移量=偏移量+data.length。
- 当一个 chunk 满了以后，再创建一个 chunk。
- 所有操作 lock free，基于 CMS 原语。

优势：

- KeyValue 原始数据在 minor gc 时被销毁。
- 数据存放在 2m 大小的 chunk 中，chunk 归属于 memstore。
- flush 时，只需要释放多个 2m 的 chunks，chunk 未满也强制释放，从而为 Heap 腾出了多个 2M 大小的内存区间，减少碎片密集程度。

5）开启 MSLAB

```
hbase.hregion.memstore.mslab.enabled=true // 开启 MSALB
hbase.hregion.memstore.mslab.chunksize=2m // chunk 的大小，越大内存连续性越好，但内存平均利用率会降低
hbase.hregion.memstore.mslab.max.allocation=256K // 通过 MSLAB 分配的对象不能超过 256K，否则直接在 Heap 上分配，256K 够大了
```

1.4 优化HBase客户端

- **AutoFlush**

将 HTable 的 setAutoFlush 设为 false，可以支持客户端批量更新。即当 Put 填满客户端 flush 缓存时，才发送到服务端。

默认是 true。

- **Scan Caching**

scanner 一次缓存多少数据来 scan（从服务端一次抓多少数据回来 scan）。

默认值是 1，一次只取一条。

- **Scan Attribute Selection**

scan 时建议指定需要的 Column Family，减少通信量，否则 scan 操作默认会返回整个 row

的所有数据（所有 Coulmn Family）。

- **Close ResultScanners**

通过 scan 取完数据后，记得要关闭 ResultScanner，否则 RegionServer 可能会出现问题（对应的 Server 资源无法释放）。

- **Optimal Loading of Row Keys**

当你 scan 一张表的时候，返回结果只需要 row key（不需要 CF, qualifier,values,timestaps）时，你可以在 scan 实例中添加一个 filterList，并设置 MUST_PASS_ALL 操作，filterList 中 add?FirstKeyOnlyFilter 或 KeyOnlyFilter。这样可以减少网络通信量。

- **Turn off WAL on Puts**

当 Put 某些非重要数据时，你可以设置 writeToWAL(false)，来进一步提高写性能。writeToWAL(false)会在 Put 时放弃写 WAL log。风险是，当 RegionServer 宕机时，可能你刚才 Put 的那些数据会丢失，且无法恢复。

- **启用 Bloom Filter**

Bloom Filter 通过空间换时间，提高读操作性能。

1.5 优化其他方面

- **启用 LZO 压缩**

LZO对比Hbase默认的GZip，前者性能较高，后者压缩比较高，具体参见 Using LZO Compression。对于想提高HBase读写性能的开发者，采用LZO是比较好的选择。对于非常在乎存储空间的开发者，则建议保持默认。

- **不要在一张表里定义太多的 Column Family**

Hbase 目前不能良好的处理超过包含 2-3 个 CF 的表。因为某个 CF 在 flush 发生时，它邻近的 CF 也会因关联效应被触发 flush，最终导致系统产生更多 IO。

- **批量导入**

在批量导入数据到Hbase前，你可以通过预先创建regions，来平衡数据的负载。详见? Table Creation: Pre-Creating Regions

- **避免 CMS concurrent mode failure**

HBase 使用 CMS GC。默认触发 GC 的时机是当年老代内存达到 90%的时候，这个百分

比由 -XX:CMSInitiatingOccupancyFraction=N 这个参数来设置。concurrent mode failed 发生在这样一个场景：

当年老代内存达到 90%的时候，CMS 开始进行并发垃圾收集，于此同时，新生代还在迅速不断地晋升对象到年老代。当年老代 CMS 还未完成并发标记时，年老代满了，悲剧就发生了。CMS 因为没内存可用不得不暂停 mark，并触发一次 stop the world（挂起所有 jvm 线程），然后采用单线程拷贝方式清理所有垃圾对象。这个过程会非常漫长。为了避免出现 concurrent mode failed，建议让 GC 在未到 90%时，就触发。

通过设置“-XX:CMSInitiatingOccupancyFraction=N”

这个百分比， 可以这么计算。如果你的“hbase.regionserver.global.memstore.upperLimit”和“hfile.block.cache.size”加起来有 60%（默认），那么你可以设置 70-80，一般高 10%左右差不多。

2、从程序角度优化

2.1 表的设计

1）Pre-Creating Regions

默认情况下，在创建 HBase 表的时候会自动创建一个 region 分区，当导入数据的时候，所有的 HBase 客户端都向这一个 region 写数据，直到这个 region 足够大了才进行切分。一种可以加快批量写入速度的方法是通过预先创建一些空的 regions，这样当数据写入 HBase 时，会按照 region 分区情况，在集群内做数据的负载均衡。

有关预分区，详情参见：Table Creation: Pre-Creating Regions，下面是一个例子：

```
public static boolean createTable(HBaseAdmin admin, HTableDescriptor table,
        byte[][] splits) throws IOException {
    try {
        admin.createTable(table, splits);
        return true;
    } catch (TableExistsException e) {
        logger.info("table " + table.getNameAsString() + " already exists");
        // the table already exists...
        return false;
    }
}

public static byte[][] getHexSplits(String startKey, String endKey,
        int numRegions) {
    byte[][] splits = new byte[numRegions - 1][];
    BigInteger lowestKey = new BigInteger(startKey, 16);
    BigInteger highestKey = new BigInteger(endKey, 16);
```

```
        BigInteger range = highestKey.subtract(lowestKey);
        BigInteger regionIncrement = range.divide(BigInteger
                .valueOf(numRegions));
        lowestKey = lowestKey.add(regionIncrement);
        for (int i = 0; i < numRegions - 1; i++) {
            BigInteger key = lowestKey.add(regionIncrement.multiply(BigInteger
                    .valueOf(i)));
            byte[] b = String.format("%016x", key).getBytes();
            splits[i] = b;
        }
        return splits;
}
```

2）Row Key

HBase 中 row key 用来检索表中的记录，支持以下三种方式：

- 通过单个 row key 访问：即按照某个 row key 键值进行 get 操作；
- 通过 row key 的 range 进行 scan：即通过设置 startRowKey 和 endRowKey，在这个范围内进行扫描；
- 全表扫描：即直接扫描整张表中所有行记录。

在 HBase 中，row key 可以是任意字符串，最大长度 64KB，实际应用中一般为 10~100bytes，存为 byte[]字节数组，一般设计成定长的。

row key 是按照字典序存储，因此，设计 row key 时，要充分利用这个排序特点，将经常一起读取的数据存储到一块，将最近可能会被访问的数据放在一块。

举个例子：如果最近写入 HBase 表中的数据是最可能被访问的，可以考虑将时间戳作为 row key 的一部分，由于是字典序排序，所以可以使用 Long.MAX_VALUE - timestamp 作为 row key，这样能保证新写入的数据在读取时可以被快速命中。

3）Column Family

不要在一张表里定义太多的 column family。目前 Hbase 并不能很好的处理超过 2~3 个 column family 的表。因为某个 column family 在 flush 的时候，它邻近的 column family 也会因关联效应被触发 flush，最终导致系统产生更多的 I/O。感兴趣的同学可以对自己的 HBase 集群进行实际测试，从得到的测试结果数据验证一下。

4）In Memory

创建表的时候，可以通过 HColumnDescriptor.setInMemory(true)将表放到 RegionServer 的缓存中，保证在读取的时候被 cache 命中。

5）Max Version

创建表的时候，可以通过 HColumnDescriptor.setMaxVersions(int maxVersions)设置表中数据的最大版本，如果只需要保存最新版本的数据，那么可以设置 setMaxVersions(1)。

6）Time To Live

创建表的时候，可以通过 HColumnDescriptor.setTimeToLive(int timeToLive)设置表中数据的存储生命期，过期数据将自动被删除，例如如果只需要存储最近两天的数据，那么可以设置 setTimeToLive(2 * 24 * 60 * 60)。

7）Compact & Split

在 HBase 中，数据在更新时首先写入 WAL 日志(HLog)和内存(MemStore)中，MemStore 中的数据是排序的，当 MemStore 累计到一定阈值时，就会创建一个新的 MemStore，并且将老的 MemStore 添加到 flush 队列，由单独的线程 flush 到磁盘上，成为一个 StoreFile。于此同时， 系统会在 zookeeper 中记录一个 redo point，表示这个时刻之前的变更已经持久化了(minor compact)。

StoreFile 是只读的，一旦创建后就不可以再修改。因此 Hbase 的更新其实是不断追加的操作。当一个 Store 中的 StoreFile 达到一定的阈值后，就会进行一次合并(major compact)，将对同一个 key 的修改合并到一起，形成一个大的 StoreFile，当 StoreFile 的大小达到一定阈值后，又会对 StoreFile 进行分割(split)，等分为两个 StoreFile。

由于对表的更新是不断追加的，处理读请求时，需要访问 Store 中全部的 StoreFile 和 MemStore，将它们按照 row key 进行合并，由于 StoreFile 和 MemStore 都是经过排序的，并且 StoreFile 带有内存中索引，通常合并过程还是比较快的。

实际应用中，可以考虑必要时手动进行 major compact，将同一个 row key 的修改进行合并形成一个大的 StoreFile。同时，可以将 StoreFile 设置大些，减少 split 的发生。

2.2 写表操作

1）多 HTable 并发写

创建多个 HTable 客户端用于写操作，提高写数据的吞吐量，一个例子：

```
static final Configuration conf = HBaseConfiguration.create();
static final String table_log_name = “user_log”;
wTableLog = new HTable[tableN];
for (int i = 0; i < tableN; i++) {
    wTableLog[i] = new HTable(conf, table_log_name);
    wTableLog[i].setWriteBufferSize(5 * 1024 * 1024); //5MB
    wTableLog[i].setAutoFlush(false);
}
```

2） HTable 参数设置

- **Auto Flush**

通过调用 HTable.setAutoFlush(false)方法可以将 HTable 写客户端的自动 flush 关闭，这样可以批量写入数据到 HBase，而不是有一条 put 就执行一次更新，只有当 put 填满客户端写缓存时，才实际向 HBase 服务端发起写请求。默认情况下 auto flush 是开启的。

■ Write Buffer

通过调用 HTable.setWriteBufferSize(writeBufferSize)方法可以设置 HTable 客户端的写 buffer 大小，如果新设置的 buffer 小于当前写 buffer 中的数据时，buffer 将会被 flush 到服务端。其中，writeBufferSize 的单位是 byte 字节数，可以根据实际写入数据量的多少来设置该值。

■ WAL Flag

在 HBae 中，客户端向集群中的 RegionServer 提交数据时（Put/Delete 操作），首先会先写 WAL（Write Ahead Log）日志（即 HLog，一个 RegionServer 上的所有 Region 共享一个 HLog），只有当 WAL 日志写成功后，再接着写 MemStore，然后客户端被通知提交数据成功；如果写 WAL 日志失败，客户端则被通知提交失败。这样做的好处是可以做到 RegionServer 宕机后的数据恢复。

因此，对于相对不太重要的数据，可以在 Put/Delete 操作时，通过调用 Put.setWriteToWAL(false)或 Delete.setWriteToWAL(false)函数，放弃写 WAL 日志，从而提高数据写入的性能。

值得注意的是：谨慎选择关闭 WAL 日志，因为这样的话，一旦 RegionServer 宕机，Put/Delete 的数据将会无法根据 WAL 日志进行恢复。

3）批量写

通过调用 HTable.put(Put)方法可以将一个指定的 row key 记录写入 HBase，同样 HBase 提供了另一个方法：通过调用 HTable.put(List<Put>)方法可以将指定的 row key 列表，批量写入多行记录，这样做的好处是批量执行，只需要一次网络 I/O 开销，这对于对数据实时性要求高，网络传输 RTT 高的情景下可能带来明显的性能提升。

4）多线程并发写

在客户端开启多个 HTable 写线程，每个写线程负责一个 HTable 对象的 flush 操作，这样结合定时 flush 和写 buffer（writeBufferSize），可以既保证在数据量小的时候，数据可以在较短时间内被 flush（如 1 秒内），同时又保证在数据量大的时候，写 buffer 一满就及时进行 flush。下面给个具体的例子：

```
for (int i = 0; i < threadN; i++) {
    Thread th = new Thread() {
        public void run() {
            while (true) {
                try {
                    sleep(1000); // 1 second
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                synchronized (wTableLog[i]) {
                    try {
                        wTableLog[i].flushCommits();
```

```
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }
                }
            }
        };
        th.setDaemon(true);
        th.start();
    }
```

2.3 读表操作

1）多 HTable 并发读

创建多个 HTable 客户端用于读操作，提高读数据的吞吐量，一个例子：

```
static final Configuration conf = HBaseConfiguration.create();
static final String table_log_name = “user_log”;
rTableLog = new HTable[tableN];
for (int i = 0; i < tableN; i++) {
    rTableLog[i] = new HTable(conf, table_log_name);
    rTableLog[i].setScannerCaching(50);
}
```

2）HTable 参数设置

■ Scanner Caching

通过调用 HTable.setScannerCaching(int scannerCaching)可以设置 HBase scanner 一次从服务端抓取的数据条数，默认情况下一次一条。通过将此值设置成一个合理的值，可以减少 scan 过程中 next()的时间开销，代价是 scanner 需要通过客户端的内存来维持这些被 cache 的行记录。

■ Scan Attribute Selection

scan 时指定需要的 Column Family，可以减少网络传输数据量，否则默认 scan 操作会返回整行所有 Column Family 的数据。

■ Close ResultScanner

通过 scan 取完数据后，记得要关闭 ResultScanner，否则 RegionServer 可能会出现问题（对应的 Server 资源无法释放）。

3）批量读

通过调用 HTable.get(Get)方法可以根据一个指定的 row key 获取一行记录，同样 HBase 提供了另一个方法：通过调用 HTable.get(List<Get>)方法可以根据一个指定的 row key 列表，批量获取多行记录，这样做的好处是批量执行，只需要一次网络 I/O 开销，这对于对数据实

时性要求高而且网络传输 RTT 高的情景下可能带来明显的性能提升。

4）多线程并发读

在客户端开启多个 HTable 读线程，每个读线程负责通过 HTable 对象进行 get 操作。下面是一个多线程并发读取 HBase，获取店铺一天内各分钟 PV 值的例子：

```
public class DataReaderServer {
    // 获取店铺一天内各分钟 PV 值的入口函数
    public static ConcurrentHashMap<String, String> getUnitMinutePV(
            long uid, long startStamp, long endStamp) {
        long min = startStamp;
        int count = (int) ((endStamp - startStamp) / (60 * 1000));
        List<String> lst = new ArrayList<String>();
        for (int i = 0; i <= count; i++) {
            min = startStamp + i * 60 * 1000;
            lst.add(uid + "_" + min);
        }
        return parallelBatchMinutePV(lst);
    }

    // 多线程并发查询，获取分钟 PV 值
    private static ConcurrentHashMap<String, String> parallelBatchMinutePV(List<String> lstKeys){
        ConcurrentHashMap<String, String> hashRet = new ConcurrentHashMap<String, String>();
        int parallel = 3;
        List<List<String>> lstBatchKeys    = null;
        if (lstKeys.size() < parallel ){
            lstBatchKeys    = new ArrayList<List<String>>(1);
            lstBatchKeys.add(lstKeys);
        }
        else{
            lstBatchKeys    = new ArrayList<List<String>>(parallel);
            for(int i = 0; i < parallel; i++    ){
                List<String> lst = new ArrayList<String>();
                lstBatchKeys.add(lst);
            }

            for(int i = 0 ; i < lstKeys.size() ; i ++ ){
                lstBatchKeys.get(i%parallel).add(lstKeys.get(i));
            }
        }

        List<Future< ConcurrentHashMap<String, String> >> futures =
                new ArrayList<Future< ConcurrentHashMap<String, String> >>(5);
```

```
ThreadFactoryBuilder builder = new ThreadFactoryBuilder();
builder.setNameFormat("ParallelBatchQuery");
ThreadFactory factory = builder.build();
ThreadPoolExecutor executor =
(ThreadPoolExecutor) Executors.newFixedThreadPool(lstBatchKeys.size(), factory);

for(List<String> keys : lstBatchKeys){
      Callable< ConcurrentHashMap<String, String> > callable =
                                          new BatchMinutePVCallable(keys);
      FutureTask< ConcurrentHashMap<String, String> > future =
        (FutureTask< ConcurrentHashMap<String, String> >) executor.submit(callable);
      futures.add(future);
}
executor.shutdown();

// Wait for all the tasks to finish
try {
   boolean stillRunning = !executor.awaitTermination(TimeUnit.MILLISECONDS);
   if (stillRunning) {
      try {
            executor.shutdownNow();
      } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
      }
   }
} catch (InterruptedException e) {
   try {
         Thread.currentThread().interrupt();
   } catch (Exception e1) {
      // TODO Auto-generated catch block
      e1.printStackTrace();
   }
}

// Look for any exception
for (Future f : futures) {
   try {
         if(f.get() != null)
         {
               hashRet.putAll((ConcurrentHashMap<String, String>)f.get());
         }
   } catch (InterruptedException e) {
      try {
```

```
                    Thread.currentThread().interrupt();
                } catch (Exception e1) {
                    // TODO Auto-generated catch block
                    e1.printStackTrace();
                }
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
        }
        return hashRet;
    }

    // 一个线程批量查询，获取分钟 PV 值
    protected static ConcurrentHashMap<String, String> getBatchMinutePV(
            List<String> lstKeys) {
        ConcurrentHashMap<String, String> hashRet = null;
        List<Get> lstGet = new ArrayList<Get>();
        String[] splitValue = null;
        for (String s : lstKeys) {
            splitValue = s.split("_");
            long uid = Long.parseLong(splitValue[0]);
            long min = Long.parseLong(splitValue[1]);
            byte[] key = new byte[16];
            Bytes.putLong(key, 0, uid);
            Bytes.putLong(key, 8, min);
            Get g = new Get(key);
            g.addFamily(fp);
            lstGet.add(g);
        }
        Result[] res = null;
        try {
            res = tableMinutePV[rand.nextInt(tableN)].get(lstGet);
        } catch (IOException e1) {
            logger.error("tableMinutePV exception, e=" + e1.getStackTrace());
        }

        if (res != null && res.length > 0) {
            hashRet = new ConcurrentHashMap<String, String>(res.length);
            for (Result re : res) {
                if (re != null && !re.isEmpty()) {
                    try {
                        byte[] key = re.getRow();
                        byte[] value = re.getValue(fp, cp);
                        if (key != null && value != null) {
```

```
                                    hashRet.put(String.valueOf(Bytes.toLong(key,
                                            Bytes.SIZEOF_LONG)), String
                                            .valueOf(Bytes.toLong(value)));
                                }
                            } catch (Exception e2) {
                                logger.error(e2.getStackTrace());
                            }
                        }
                    }
                }
                return hashRet;
        }
}

// 调用接口类，实现 Callable 接口
class BatchMinutePVCallable implements
            Callable<ConcurrentHashMap<String, String>> {
        private List<String> keys;
        public BatchMinutePVCallable(List<String> lstKeys) {
                this.keys = lstKeys;
        }
        public ConcurrentHashMap<String, String> call() throws Exception {
                return DataReadServer.getBatchMinutePV(keys);
        }
}
```

5）缓存查询结果

对于频繁查询 HBase 的应用场景，可以考虑在应用程序中做缓存，当有新的查询请求时，首先在缓存中查找，如果存在则直接返回，不再查询 HBase；否则对 HBase 发起读请求查询，然后在应用程序中将查询结果缓存起来。至于缓存的替换策略，可以考虑 LRU 等常用的策略。

6） Blockcache

HBase 上 Regionserver 的内存分为两个部分，一部分作为 Memstore，主要用来写；另外一部分作为 BlockCache，主要用于读。

写请求会先写入 Memstore，Regionserver 会给每个 region 提供一个 Memstore，当 Memstore 满 64MB 以后，会启动 flush 刷新到磁盘。当 Memstore 的总大小超过限制时（heapsize * hbase.regionserver.global.memstore.upperLimit * 0.9），会强行启动 flush 进程，从最大的 Memstore 开始 flush 直到低于限制。

读请求先到 Memstore 中查数据，查不到就到 BlockCache 中查，再查不到就会到磁盘上读，并把读的结果放入 BlockCache。由于 BlockCache 采用的是 LRU 策略，因此 BlockCache 达到上限(heapsize * hfile.block.cache.size * 0.85)后，会启动淘汰机制，淘汰掉最老的一批数

据。

一个 Regionserver 上有一个 BlockCache 和 N 个 Memstore，它们的大小之和不能大于等于 heapsize * 0.8，否则 HBase 不能启动。默认 BlockCache 为 0.2，而 Memstore 为 0.4。**对于注重读响应时间的系统，可以将 BlockCache 设大些，比如设置 BlockCache=0.4，Memstore=0.39，以加大缓存的命中率。**

2.4 数据计算

1）服务端计算

Coprocessor 运行于 HBase RegionServer 服务端，各个 Regions 保持对与其相关的 coprocessor 实现类的引用，coprocessor 类可以通过 RegionServer 上 classpath 中的本地 jar 或 HDFS 的 classloader 进行加载。

目前，已提供有几种 coprocessor：

- Coprocessor：提供对于 region 管理的钩子；
- RegionObserver：提供用于从客户端监控表相关操作的钩子，例如表的 get/put/scan/delete 等；
- Endpoint：提供可以在 region 上执行任意函数的命令触发器。一个使用例子是 RegionServer 端的列聚合，这里有代码示例。

以上只是有关 coprocessor 的一些基本介绍，本人没有对其实际使用的经验，对它的可用性和性能数据不得而知。感兴趣的同学可以尝试一下，欢迎讨论。

2）写端计算

■ 计数

HBase 本身可以看作是一个可以水平扩展的 Key-Value 存储系统，但是其本身的计算能力有限（Coprocessor 可以提供一定的服务端计算），因此，使用 HBase 时，往往需要从写端或者读端进行计算，然后将最终的计算结果返回给调用者。举两个简单的例子：

- **PV 计算**：通过在 HBase 写端内存中，累加计数，维护 PV 值的更新，同时为了做到持久化，定期（如 1 秒）将 PV 计算结果同步到 HBase 中，这样查询端最多会有 1 秒钟的延迟，能看到秒级延迟的 PV 结果。
- **分钟 PV 计算**：与上面提到的 PV 计算方法相结合，每分钟将当前的累计 PV 值，按照 rowkey + minute 作为新的 rowkey 写入 HBase 中，然后在查询端通过 scan 得到当天各个分钟以前的累计 PV 值，然后顺次将前后两分钟的累计 PV 值相减，就得到了当前一分钟内的 PV 值，从而最终也就得到当天各个分钟内的 PV 值。

■ 去重

对于 UV 的计算，就是个去重计算的例子。分两种情况：

- 如果内存可以容纳，那么可以在 Hash 表中维护所有已经存在的 UV 标识，每当新来一个标识时，通过快速查找 Hash 确定是否是一个新的 UV，若是则 UV 值加 1，否则 UV 值不变。另外，为了做到持久化或提供给查询接口使用，可以定期（如 1 秒）将 UV 计算结果同步到 HBase 中。
- 如果内存不能容纳，可以考虑采用 Bloom Filter 来实现，从而尽可能的减少内存的

占用情况。除了 UV 的计算外，判断 URL 是否存在也是个典型的应用场景。

3）读端计算

如果对于响应时间要求比较苛刻的情况（如单次 http 请求要在毫秒级时间内返回），个人觉得读端不宜做过多复杂的计算逻辑，尽量做到读端功能单一化：即从 HBase RegionServer 读到数据（scan 或 get 方式）后，按照数据格式进行简单的拼接，直接返回给前端使用。当然，如果对于响应时间要求一般，或者业务特点需要，也可以在读端进行一些计算逻辑。

2.5 优化总结

作为一个 Key-Value 存储系统，HBase 并不是万能的，它有自己独特的地方。因此，基于它来做应用时，我们往往需要从多方面进行优化改进（表设计、读表操作、写表操作、数据计算等），有时甚至还需要从系统级对 HBase 进行配置调优，更甚至可以对 HBase 本身进行优化。这属于不同的层次范畴。

总之，概括来讲，对系统进行优化时，首先定位到影响你的程序运行性能的瓶颈之处，然后有的放矢进行针对行的优化。如果优化后满足你的期望，那么就可以停止优化；否则继续寻找新的瓶颈之处，开始新的优化，直到满足性能要求。

3、HBase性能深入分析

对于 Bigtable 类型的分布式数据库应用来说，用户往往会对其性能状况有极大的兴趣，这其中又对实时数据插入性能更为关注。HBase 作为 Bigtable 的一个实现，在这方面的性能会如何呢？这就需要通过测试数据来说话了。

数据插入性能测试的设计场景是这样的，取随机值的 Rowkey 长度为 2000 字节，固定值的 Value 长度为 4000 字节，由于单行 Row 插入速度太快，系统统计精度不够，所以将插入 500 行 Row 做一次耗时统计。

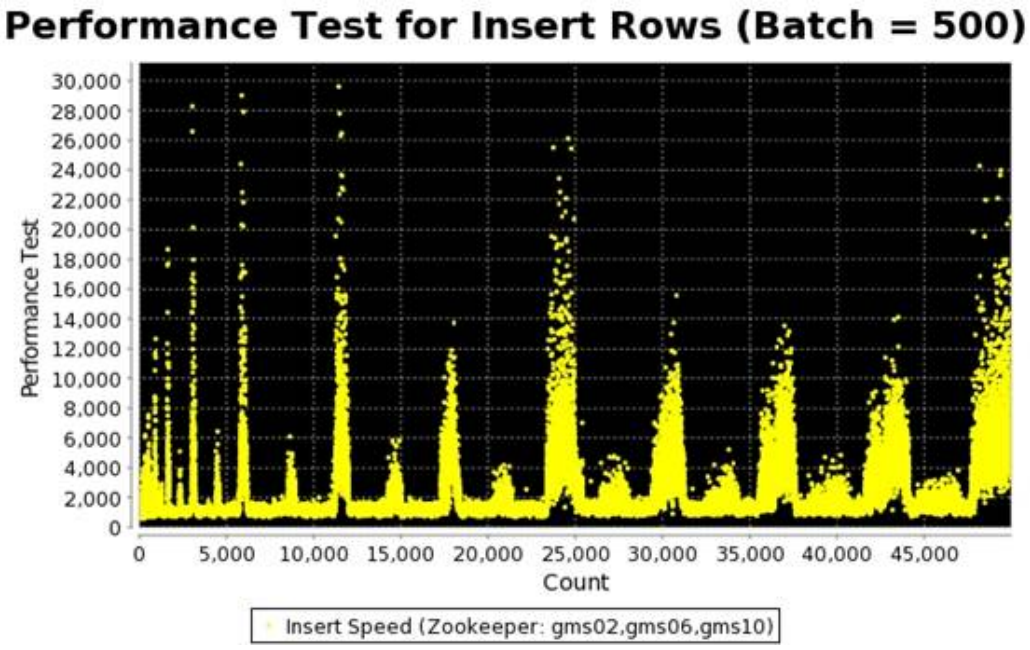
这里要对 HBase 的特点做个说明，首先是 Rowkey 值为何取随机数，这是因为 HBase 是对 Rowkey 进行排序的，随机 Rowkey 将被分配到不同的 region 上，这样才能发挥出分布式数据库的性能优点。而 Value 对于 HBase 来说不会进行任何解析，其数据是否变化，对性能是不应该有任何影响的。同时为了简单起见，所有的数据都将只插入到一个表格的同一个 Column 中。

在测试之初，需要对集群进行调优，关闭可能大量耗费内存、带宽以及 CPU 的服务，例如 Apache 的 Http 服务。保持集群的宁静度。此外，为了保证测试不受干扰，Hbase 的集群系统需要被独立，以保证不与 HDFS 所在的 Hadoop 集群有所交叉。

那么做好一切准备，就开始进行数据灌入，客户端从 Zookeeper 上查询到 Regionserver 的地址后，开始源源不断的向 Hbase 的 Regionserver 上喂入 Row。

这里，我写了一个通过 JFreeChart 来实时生成图片的程序，每 3 分钟，喂数据的客户端

会将获取到的耗时统计打印在一张十字坐标图中，这些图又被保存在制定的 web 站点中，并通过 http 服务展示出来。在通过长时间不间断的测试后，我得到了如下图形：



这个图形非常有特点，好似一条直线上，每隔一段时间就会泛起一个波浪，且两个高峰之间必有一个较矮的波浪。高峰的间隔则呈现出越来越大的趋势。而较矮的波浪恰好处于两高峰的中间位置。

为了解释这个现象，我对 HDFS 上 Hbase 所在的主目录下文件，以及被插入表格的 region 情况进行了实时监控，以期发现这些波浪上发生了什么事情。

回溯到客户端喂入数据的开始阶段，创建表格，在 HDFS 上便被创建了一个与表格同名的目录，该目录下将出现第一个 region，region 中会以 family 名创建一个目录，这个目录下才存在记录具体数据的文件。同时在该表表名目录下，还会生成一个“compaction.dir”目录，该目录将在 family 名目录下 region 文件超过指定数目时用于合并 region。

当第一个 region 目录出现的时候，内存中最初被写入的数据将被保存到这个文件中，这个间隔是由选项“hbase.hregion.memstore.flush.size”决定的，默认是 64MB，该 region 所在的 Regionserver 的内存中一旦有超过 64MB 的数据的时候，就将被写入到 region 文件中。这个文件将不断增殖，直到超过由“hbase.hregion.max.filesize”决定的文件大小时（默认是 256MB，此时加上内存刷入的数据，实际最大可能到 256+64M），该 region 将被执行 split，立即被一切为二，其过程是在该目录下创建一个名为“.splits”的目录作为标记，然后由 Regionserver 将文件信息读取进来，分别写入到两个新的 region 目录中，最后再将老的 region 删除。这里的标记目录“.splits”将避免在 split 过程中发生其他操作，起到类似于多线程安全的锁功能。在新的 region 中，从老的 region 中切分出的数据独立为一个文件并不再接受新的数据（该文件大小超过了 64M，最大可达到（256+64）/2=160MB），内存中新的数据将被保存到一个重新创建的文件中，该文件大小将为 64MB。内存每刷新一次，region 所在的目录下就将增加一个 64M 的文件，直到总文件数超过由

“hbase.hstore.compactionThreshold”指定的数量时（默认为 3），compaction 过程就将被触发了。在上述值为 3 时，此时该 region 目录下，实际文件数只有两个，还有额外的一个正处于内存中将要被刷入到磁盘的过程中。Compaction 过程是 Hbase 的一个大动作，Hbase 不仅要将这些文件转移到“compaction.dir”目录进行压缩，而且在压缩后的文件超过 256MB 时，还必须立即进行 split 动作。这一系列行为在 HDFS 上可谓是翻山倒海，影响颇大。待 Compaction 结束之后，后续的 split 依然会持续进行一小段时间，直到所有的 region 都被切割分配完毕，Hbase 才会恢复平静并等待下一次数据从内存写入到 HDFS 的到来。

理解了上述过程，则必然对 HBase 的数据插入性能为何是上图所示的曲线的原因一目了然。与 X 轴几乎平行的直线，表明数据正在被写入 HBase 的 Regionserver 所在机器的内存中。而较低的波峰意味着 Regionserver 正在将内存写入到 HDFS 上，较高的波峰意味着 Regionserver 不仅正在将内存刷入到 HDFS，而且还在执行 Compaction 和 Split 两种操作。如果调整“hbase.hstore.compactionThreshold”的值为一个较大的数量，例如改成 5，可以预见，在每两个高峰之间必然会等间隔的出现三次较低的波峰，并可预见到，高峰的高度将远超过上述值为 3 时的高峰高度（因为 Compaction 的工作更为艰巨）。由于 region 数量由少到多，而我们插入的 Row 的 Rowkey 是随机的，因此每一个 region 中的数据都会均匀的增加，同一段时间插入的数据将被分布到越来越多的 region 上，因此波峰之间的间隔时间也将会越来越长。

再次理解上述论述，我们可以推断出 Hbase 的数据插入性能实际上应该被分为三种情况，即直线状态、低峰状态和高峰状态。在这三种情况下得到的性能数据才是最终 Hbase 数据插入性能的真实描述。那么提供给用户的数据该是采取哪一个呢？我认为直线状态由于其所占时间会较长，尤其在用户写入数据的速度也许并不是那么快的情况下，所以这个状态下得到的性能数据结果更应该提供给用户。

4、HBase在淘宝应用及优化

4.1 前言

hbase 是从 hadoop 中 分离出来的 apache 顶级开源项目。由于它很好地用 java 实现了 google 的 bigtable 系统大部分特性，因此在数据量猛增的今天非常受到欢 迎。对于淘宝而言，随着市场规模的扩大，产品与技术的发展，业务数据量越来越大，对海量数据的高效插入和读取变得越来越重要。由于淘宝拥有也许是国内最大 的单一 hadoop 集群(云梯)，因此对 hadoop 系列的产品有比较深入的了解，也就自然希望使用 hbase 来做这样一种海量数据读写服务。本节内容将对淘宝最近一年来在 online 应用上使用和优化 hbase 的情况做一次小结。

4.2 原因

为什么要使用 hbase？

淘宝在 2011 年之前所有的后端持久化存储基本上都是在 mysql 上进行的(不排除少量

oracle/bdb/tair/mongdb 等)，mysql 由于开源，并且生态系统良好，本身拥有分库分表等多种解决方案，因此很长一段时间内都满足淘宝大量业务的需求。

但是由于业务的多样化发展，有越来越多的业务系统的需求开始发生了变化。一般来说有以下几类变化：

a) 数据量变得越来越多，事实上现在淘宝几乎任何一个与用户相关的在线业务的数据量都在亿级别，每日系统调用次数从亿到百亿都有，且历史数据不能轻易删除。这需要有一个海量分布式文件系统，能对 TB 级甚至 PB 级别的数据提供在线服务
b) 数据量的增长很快且不一定能准确预计，大多数应用系统从上线起在一段时间内数据量都呈很快的上升趋势，因此从成本的角度考虑对系统水平扩展能力有比较强烈的需求，且不希望存在单点制约
c) 只需要简单的 kv 读取，没有复杂的 join 等需求。但对系统的并发能力以及吞吐量、响应延时有非常高的需求，并且希望系统能够保持强一致性
d) 通常系统的写入非常频繁，尤其是大量系统依赖于实时的日志分析
e) 希望能够快速读取批量数据
f) schema 灵活多变，可能经常更新列属性或新增列
g) 希望能够方便使用，有良好且语义清晰的 java 接口

以上需求综合在一起，我们认为 hbase 是一种比较适合的选择。首先它的数据由 hdfs 天然地做了数据冗余，云梯三年的稳定运行，数据 100%可靠 已经证明了 hdfs 集群的安全性，以及服务于海量数据的能力。其次 hbase 本身的数据读写服务没有单点的限制，服务能力可以随服务器的增长而线性增长， 达到几十上百台的规模。LSM-Tree 模式的设计让 hbase 的写入性能非常良好，单次写入通常在 1-3ms 内即可响应完成，且性能不随数据量的增长而 下降。region（相当于数据库的分表）可以 ms 级动态的切分和移动，保证了负载均衡性。由于 hbase 上的数据模型是按 rowkey 排序存储的，而读 取时会一次读取连续的整块数据做为 cache，因此良好的 rowkey 设计可以让批量读取变得十分容易，甚至只需要 1 次 io 就能获取几十上百条用户想要的 数据。最后，淘宝大部分工程师是 java 背景的同学，因此 hbase 的 api 对于他们来说非常容易上手，培训成本相对较低。

当然也必须指出，在大数据量的背景下银弹是不存在的，hbase 本身也有不适合的场景。比如，索引只支持主索引（或看成主组合索引），又比如服务是 单点的，单台机器宕机后在 master 恢复它期间它所负责的部分数据将无法服务等。这就要求在选型上需要对自己的应用系统有足够了解。

4.3 应用情况

我们从 2011 年 3 月开始研究 hbase 如何用于在线服务。尽管之前在一淘搜索中已经有了几十节点的离线服务。这是因为 hbase 早期版本的目标就 是一个海量数据中的离线服务。2009 年 9 月发布的 0.20.0 版本是一个里程碑，online 应用正式成为了 hbase 的目标，为此 hbase 引入了 zookeeper 来做为 backupmaster 以及 regionserver 的管理。2011 年 1 月 0.90.0 版本是另一个里程碑，基本上我们今天 看到的各大网站，如 facebook/ebay/yahoo 内所使用于生产的 hbase 都是基于这一个版本(fb 所采用的 0.89 版本结构与 0.90.x 相近)。bloomfilter 等诸多属性加入了进来，性能也有极大提升。基于此，淘宝也选用了 0.90.x 分支作为线上版本的基

础。

第一个上线的应用是数据魔方中的 prom。prom 原先是基于 redis 构建的，因为数据量持续增大以及需求的变化，因此我们用 hbase 重构了它 的存储层。准确的说 prom 更适合 0.92 版本的 hbase，因为它不仅需要高速的在线读写，更需要 count/group by 等复杂应用。但由于当时 0.92 版本尚未成熟，因此我们自己单独实现了 coprocessor。prom 的数据导入是来源于云梯，因此我们每天晚上花 半个小时将数据从云梯上写入 hbase 所在的 hdfs，然后在 web 层做了一个 client 转发。经过一个月的数据比对，确认了速度比之 redis 并未有 明显下降，以及数据的准确性，因此得以顺利上线。

第二个上线的应用是 TimeTunnel，TimeTunnel 是一个高效的、可靠的、可扩展的实时数据传输平台，广泛应用于实时日志收集、数据实 时监控、广告效果实时反馈、数据库实时同步等领域。它与 prom 相比的特点是增加了在线写。动态的数据增加使 hbase 上 compact/balance /split/recovery 等诸多特性受到了极大的挑战。TT 的写入量大约一天 20TB，读的量约为此的 1.5 倍，我们为此准备了 20 台 regionserver 的集群，当然底层的 hdfs 是公用的，数量更为庞大（下文会提到）。每天 TT 会为不同的业务在 hbase 上建不同的表，然后往该 表上写入数据，即使我们将 region 的大小上限设为 1GB，最大的几个业务也会达到数千个 region 这样的规模，可以说每一分钟都会有数次 split。在 TT 的上线过程中，我们修复了 hbase 很多关于 split 方面的 bug，有好几个 commit 到了 hbase 社区，同时也将社区一些最新 的 patch 打在了我们的版本上。split 相关的 bug 应该说是 hbase 中会导致数据丢失最大的风险之一，这一点对于每个想使用 hbase 的开发者来 说必须牢记。hbase 由于采用了 LSM-Tree 模型，从架构原理上来说数据几乎没有丢失的可能，但是在实际使用中不小心谨慎就有丢失风险。原因后面会 单独强调。TT 在预发过程中我们分别因为 Meta 表损坏以及 split 方面的 bug 曾经丢失过数据，因此也单独写了 meta 表恢复工具，确保今后不发生类 似问题(hbase-0.90.5 以后的版本都增加了类似工具)。另外，由于我们存放 TT 的机房并不稳定，发生过很多次宕机事故，甚至发生过假死现象。因 此我们也着手修改了一些 patch，以提高宕机恢复时间，以及增强了监控的强度。

CTU 以及会员中心项目是两个对在线要求比较高的项目，在这两个项目中我们特别对 hbase 的慢响应问题进行了研究。hbase 的慢响应现在一般归 纳为四类原因：网络原因、gc 问题、命中率以及 client 的反序列化问题。我们现在对它们做了一些解决方案(后面会有介绍)，以更好地对慢响应有控制 力。

和 Facebook 类似，我们也使用了 hbase 做为实时计算类项目的存储层。目前对内部已经上线了部分实时项目，比如实时页面点击系 统，galaxy 实时交易推荐以及直播间等内部项目，用户则是散布到公司内各部门的运营小二们。与 facebook 的 puma 不同的是淘宝使用了多种方式 做实时计算层，比如 galaxy 是使用类似 affa 的 actor 模式处理交易数据，同时关联商品表等维度表计算排行(TopN)，而实时页面点击系统则是 基于 twitter 开源的 storm 进行开发，后台通过 TT 获取实时的日志数据，计算流将中间结果以及动态维表持久化到 hbase 上，比如我们将 rowkey 设计为 url+userid，并读出实时的数据，从而实现实时计算各个维度上的 uv。

最后要特别提一下历史交易订单项目。这个项目实际上也是一个重构项目，目的是从以前的 solr+bdb 的方案上迁移到 hbase 上来。由于它关系到 已买到页面，用户使用频率非常

高，重要程度接近核心应用，对数据丢失以及服务中断是零容忍。它对 compact 做了优化，避免大数据量的 compact 在 服务时间内发生。新增了定制的 filter 来实现分页查询，rowkey 上对应用进行了巧妙的设计以避免了冗余数据的传输以及 90%以上的读转化成了顺序 读。目前该集群存储了超过百亿的订单数据以及数千亿的索引数据，线上故障率为 0。

随着业务的发展，目前我们定制的 hbase 集群已经应用到了线上超过二十个应用，数百台服务器上。包括淘宝首页的商品实时推荐、广泛用于卖家的实时量子统计等应用，并且还有继续增多以及向核心应用靠近的趋势。

4.4 部署、运维和监控

Facebook 之前曾经透露过 Facebook 的 hbase 架构，可以说是非常不错的。如他们将 message 服务的 hbase 集群按用户分为数 个集群，每个集群 100 台服务器，拥有一台 namenode 以及分为 5 个机架，每个机架上一台 zookeeper。可以说对于大数据量的服务这是一种优良的架构。对于淘宝来说，由于数据量远没有那么大，应用也没有那么核心，因此我们采用公用 hdfs 以及 zookeeper 集群的架构。每个 hdfs 集群尽量 不超过 100 台规模（这是为了尽量限制 namenode 单点问题）。在其上架设数个 hbase 集群，每个集群一个 master 以及一个 backupmaster。公用 hdfs 的好处是可以尽量减少 compact 的影响，以及均摊掉硬盘的成本，因为总有集群对磁盘空间要求高，也总有集群对 磁盘空间要求低，混合在一起用从成本上是比较合算的。zookeeper 集群公用，每个 hbase 集群在 zk 上分属不同的根节点。通过 zk 的权限机制来保 证 hbase 集群的相互独立。zk 的公用原因则仅仅是为了运维方便。

由于是在线应用，运维和监控就变得更加重要，由于之前的经验接近 0，因此很难招到专门的 hbase 运维人员。我们的开发团队和运维团队从一开始就很重视该问题，很早就开始自行培养。以下讲一些我们的运维和监控经验。

我们定制的 hbase 很重要的一部分功能就是增加监控。hbase 本身可以发送 ganglia 监控数据，只是监控项远远不够，并且 ganglia 的 展示方式并不直观和突出。因此一方面我们在代码中侵入式地增加了很多监控点，比如 compact/split/balance/flush 队列以及各个阶 段的耗时、读写各个阶段的响应时间、读写次数、region 的 open/close，以及具体到表和 region 级别的读写次数等等。仍然将它们通过 socket 的方式发送到 ganglia 中，ganglia 会把它们记录到 rrd 文件中，rrd 文件的特点是历史数据的精度会越来越低，因此我们自己编写 程序从 rrd 中读出相应的数据并持久化到其它地方，然后自己用 js 实现了一套监控界面，将我们关心的数据以趋势图、饼图等各种方式重点汇总和显示出来，并 且可以无精度损失地查看任意历史数据。在显示的同时会把部分非常重要的数据，如读写次数、响应时间等写入数据库，实现波动报警等自定义的报警。经过以上措 施，保证了我们总是能先于用户发现集群的问题并及时修复。我们利用 redis 高效的排序算法实时地将每个 region 的读写次数进行排序，能够在高负载的 情况下找到具体请求次数排名较高的那些 region，并把它们移到空闲的 regionserver 上去。在高峰期我们能对上百台机器的数十万个 region 进行实时排序。

为了隔离应用的影响，我们在代码层面实现了可以检查不同 client 过来的连接，并且切断某些 client 的连接，以在发生故障时，将故障隔离在某个应用内部而不扩大化。mapreduce 的应用也会控制在低峰期运行，比如在白天我们会关闭 jobtracker 等。

此外，为了保障服务从结果上的可用，我们也会定期跑读写测试、建表测试、hbck 等命令。hbck 是一个非常有用的工具，不过要注意它也是一个很重 的工操作，因此尽量减少 hbck 的调用次数，尽量不要并行运行 hbck 服务。在 0.90.4 以前的 hbck 会有一些机率使 hbase 宕机。另外为了确保 hdfs 的安全性，需要定期运行 fsck 等以检查 hdfs 的状态，如 block 的 replica 数量等。

我们会每天根踪所有线上服务器的日志，将错误日志全部找出来并且邮件给开发人员，以查明每一次 error 以上的问题原因和 fix。直至错误降低为 0。另外 每一次的 hbck 结果如果有问题也会邮件给开发人员以处理掉。尽管并不是每一次 error 都会引发问题，甚至大部分 error 都只是分布式系统中的正常现 象，但明白它们问题的原因是非常重要的。

4.5 测试与发布

因为是未知的系统，我们从一开始就非常注重测试。测试从一开始就分为性能测试和功能测试。性能测试主要是注意基准测试，分很多场景，比如不同混合读 写比例，不同 k/v 大小，不同列族数，不同命中率，是否做 presharding 等等。每次运行都会持续数小时以得到准确的结果。因此我们写了一套自动化 系统，从 web 上选择不同的场景，后台会自动将测试参数传到各台服务器上去执行。由于是测试分布式系统，因此 client 也必须是分布式的。

我们判断测试是否准确的依据是同一个场景跑多次，是否数据，以及运行曲线达到 99%以上的重合度，这个工作非常烦琐，以至于消耗了很多时间，但后来 的事实证明它非常有意义。因为我们对它建立了 100%的信任，这非常重要，比如后期我们的改进哪怕只提高 2%的性能也能被准确捕捉到，又比如某次代码修改 使 compact 队列曲线有了一些起伏而被我们看到，从而找出了程序的 bug，等等。

功能测试上则主要是接口测试和异常测试。接口测试一般作用不是很明显，因为 hbase 本身的单元测试已经使这部分被覆盖到了。但异常测试非常重要， 我们绝大部分 bug 修改都是在异常测试中发现的，这帮助我们去掉了很多生产环境中可能存在的不稳定因素，我们也提交了十几个相应的 patch 到社区，并受 到了重视和 commit。分布式系统设计的难点和复杂度都在异常处理上，我们必须认为系统在通讯的任何时候都是不可靠的。某些难以复现的问题我们会通过查 看代码大体定位到问题以后，在代码层面强行抛出异常来复现它。事实证明这非常有用。

为了方便和快速定位问题，我们设计了一套日志收集和处理的程序，以方便地从每台服务器上抓取相应的日志并按一定规律汇总。这非常重要，避免浪费大量的时间到登录不同的服务器以寻找一个 bug 的线索。

由于 hbase 社区在不停发展，以及线上或测试环境发现的新的 bug，我们需要制定一套有规律的发布模式。它既要避免频繁的发布引起的不稳定，又要 避免长期不发布导致生产版本离开发版本越来越远或是隐藏的 bug 爆发。我们强行规定每两周从内部 trunk 上 release 一个版本，该版本必须通过所有 的测试包括回归测试，并且在 release 后在一个小型的集群上 24 小时不受甘扰不停地运行。每个月会有一次发布，发布时采用最新 release 的版本， 并且将现有的集群按重要性分级发布，以确保重要应用不受新版本的潜在 bug 影响。事实证明自从我们引入这套发布机制后，由发布带来的不稳定因素大大下降 了，并且线上版本也能

保持不落后太多。

4.6 改进和优化

Facebook是一家非常值得尊敬的公司，他们毫无保留地对外公布了对hbase的所有改造，并且将他们内部实际使用的版本开源到了社区。 facebook 线上应用的一个重要特点是他们关闭了 split，以降低 split 带来的风险。与 facebook 不同，淘宝的业务数据量相对没有如此庞 大，并且由于应用类型非常丰富，我们并们并没有要求用户强行选择关闭 split，而是尽量去修改 split 中可能存在的 bug。到目前为止，虽然我们并不 能说完全解决了这个问题，但是从 0.90.2 中暴露出来的诸多跟 split 以及宕机相关的可能引发的 bug 我们的测试环境上已经被修复到接近了 0，也为社 区提交了 10 数个稳定性相关的 patch，比较重要的有以下几个：

- https://issues.apache.org/jira/browse/HBASE-4562
- https://issues.apache.org/jira/browse/HBASE-4563
- https://issues.apache.org/jira/browse/HBASE-5152
- https://issues.apache.org/jira/browse/HBASE-5100
- https://issues.apache.org/jira/browse/HBASE-4880
- https://issues.apache.org/jira/browse/HBASE-4878
- https://issues.apache.org/jira/browse/HBASE-4899

还有其它一些，我们主要将patch提交到0.92版本，社区会有commitor帮助我们backport回0.90版本。所以社区从 0.90.2 一直到 0.90.6 一共发布了 5 个 bugfix 版本后，0.90.6 版本其实已经比较稳定了。建议生产环境可以考虑这个版本。

split 这是一个很重的事务，它有一个严重的问题就是会修改 meta 表（当然宕机恢复时也有这个问题）。如果在此期间发生异常，很有可能 meta 表、rs 内存、master 内存以及 hdfs 上的文件会发生不一致，导致之后 region 重新分配时发生错误。其中一个错误就是有可能同一个 region 被两个以上的 regionserver 所服务，那么就可能出现这一个 region 所服务的数据会随机分别写到多台 rs 上，读取的时候也会分别读取，导致数 据丢失。想要恢复原状，必须删除掉其中一个 rs 上的 region，这就导致了不得不主动删掉数据，从而引发数据丢失。

前面说到慢响应的问题归纳为网络原因、gc 问题、命中率以及 client 的反序列化问题。网络原因一般是网络不稳定引起的，不过也有可能是 tcp 参 数设置问题，必须保证尽量减少包的延迟，如 nodelay 需要设置为 true 等，这些问题我们通过 tcpdump 等一系列工具专门定位过，证明 tcp 参数 对包的组装确实会造成慢连接。gc 要根据应用的类型来，一般在读比较多的应用中新生代不能设置得太小。命中率极大影响了响应的时间，我们会尽量将version数设为1以增加缓存的容量，良好的balance也能帮助充分应用好每台机器的命中率。我们为此设计了表级别的balance。

由于 hbase 服务是单点的，即宕机一台，则该台机器所服务的数据在恢复前是无法读写的。宕机恢复速度决定了我们服务的可用率。为此主要做了几点优 化。首先是将 zk 的宕机发现时间尽量缩短到 1 分钟，其次改进了 master 恢复日志为并行恢复，大大提高了 master 恢复日志的速度，然后我们修改了 openhandler 中可能出现的一些超时异常，以及死锁，去掉了日志中可能发生的 open…too long 等异常。原生的 hbase 在宕机恢复时有可能发生 10

几分钟甚至半小时无法重启的问题已经被修复掉了。另外，hdfs 层面我们将 socket.timeout 时间以及重试时间也缩短了，以降低 datanode 宕机引起的长时间 block 现象。

hbase 本身读写层面的优化我们目前并没有做太多的工作，唯一打的 patch 是 region 增加时写性能严重下降的问题。因为由于 hbase 本身 良好的性能，我们通过大量测试找到了各种应用场景中比较优良的参数并应用于生产环境后，都基本满足需求。不过这是我们接下来的重要工作。

4.7 将来计划

我们目前维护着淘宝内基于社区 0.90.x 而定制的 hbase 版本。接下来除继续 fix 它的 bug 外，会维护基于 0.92.x 修改的版本。之所以这 样，是因为 0.92.x 和 0.90.x 的兼容性并不是非常好，而且 0.92.x 修改掉的代码非常多，粗略统计会超过 30%。0.92 中有我们非常看重的一 些特性。

- 0.92 版本改进了 hfile 为 hfileV2，v2 版本的特点是将索引以及 bloomfilter 进行了大幅改造，以支持单个大 hfile 文 件。现有的 HFile 在文件大到一定程度时，index 会占用大量的内存，并且加载文件的速度会因此下降非常多。而如果 HFile 不增大的 话，region 就无法扩大，从而导致 region 数量非常多。这是我们想尽量避免的事。
- 0.92 版本改进了通讯层协议，在通讯层中增加了 length，这非常重要，它让我们可以写出 nio 的客户端，使反序列化不再成为影响 client 性能的地方。
- 0.92 版本增加了 coprocessor 特性，这支持了少量想要在 rs 上进行 count 等的应用。
- 还有其它很多优化，比如改进了 balance 算法、改进了 compact 算法、改进了 scan 算法、compact 变为 CF 级别、动态做 ddl 等等特性。

除了 0.92 版本外，0.94 版本以及最新的 trunk(0.96)也有很多不错的特性，0.94 是一个性能优化版本。它做了很多革命性工作，比如去掉 root 表，比如 HLog 进行压缩，replication 上支持多个 slave 集群，等等。我们自己也有一些优化，比如自行实现的二级索引、backup 策略等都会在内部版本上实现。

另外值得一提的是 hdfs 层面的优化也非常重要，hadoop-1.0.0 以及 cloudera-3u3 的改进对 hbase 非常有帮助，比如本地化 读、checksum 的改进、datanode 的 keepalive 设置、namenode 的 HA 策略等。我们有一支优秀的 hdfs 团队来支持我们的 hdfs 层面工作，比如定位以及 fix 一些 hdfs 层面的 bug,帮助提供一些 hdfs 上参数的建议，以及帮助实现 namenode 的 HA 等。最新的测试 表明，3u3 的 checksum+本地化读可以将随机读性能提升至少一倍。

我们正在做的一件有意义的事是实时监控和调整 regionserver 的负载，能够动态地将负载不足的集群上的服务器挪到负载较高的集群中，而整个过程对用户完全透明。

总的来说，我们的策略是尽量和社区合作，以推动 hbase 在整个 apache 生态链以及业界的发展，使其能更稳定地部署到更多的应用中去，以降低使用门槛以及使用成本。

参考文献

感谢以下文章的编写作者，没有你们的铺路，我或许会走得很艰难，参考不分先后，贡献同等珍贵。

【1】HBase 入门篇 3-hbase 配置文件参数设置及优化
地址：http://blog.csdn.net/a221133/article/details/6777433
【2】HBase 性能优化方法总结
地址：http://www.cnblogs.com/panfeng412/tag/
【3】HBase 性能调优
地址：http://kenwublog.com/hbase-performance-tuning
【4】HBase 在淘宝的应用和优化小结
地址：http://www.oschina.net/question/195301_41176
【5】HBase 性能深度分析
地址：http://www.blogjava.net/ivanwan/archive/2011/06/10/352071.html
【6】提升 HBase 性能的几个地方
地址：http://www.itokit.com/2011/0516/66217.html