Apache HBase 写数据流程说明

Apache HBase 是构建在 HDFS 之上的数据库，使用 HBase 我们可以随机读写存储在 HDFS 上的数据，但是我们都知道，HDFS 上的文件仅仅只支持追加（Append），其默认是不支持修改已经写好的文件。所以很多人就会问，HBase 是如何实现低延迟的读写能力呢？文本将试图介绍 HBase 写数据的过程。

其实 HBase 写数据包括 put 和 delete 操作，在 HBase 里面添加数据和更新数据其实就是一个 put 操作；而 delete
数据并不是把原有的数据立即删除，而仅仅是做一个标记操作，真实的数据会在后面的 Major Compaction 过程中删除的。我们往 HBase 里面写数据，首先是经过 HBase Client 的，然后到 RegionServer，最后数据被写到对应的 HFile 里面。

每张 HBase 的表都是由以下三种类型的服务器托管和元数据管理：

- 一个活动（Active）的 master server
- 一个备（backup） master server
- 许多 RegionServer

最终的表数据其实是由 RegionServer 管理的。由于 HBase
表的数据往往很大，这些数据进一步分成许多的 Regions，每个 RegionServer 管理一个或多个 Region。需要注意的是，因为表的数据只有 RegionServer 管理的，所以 master
节点挂掉并不会导致我们数据的丢失。

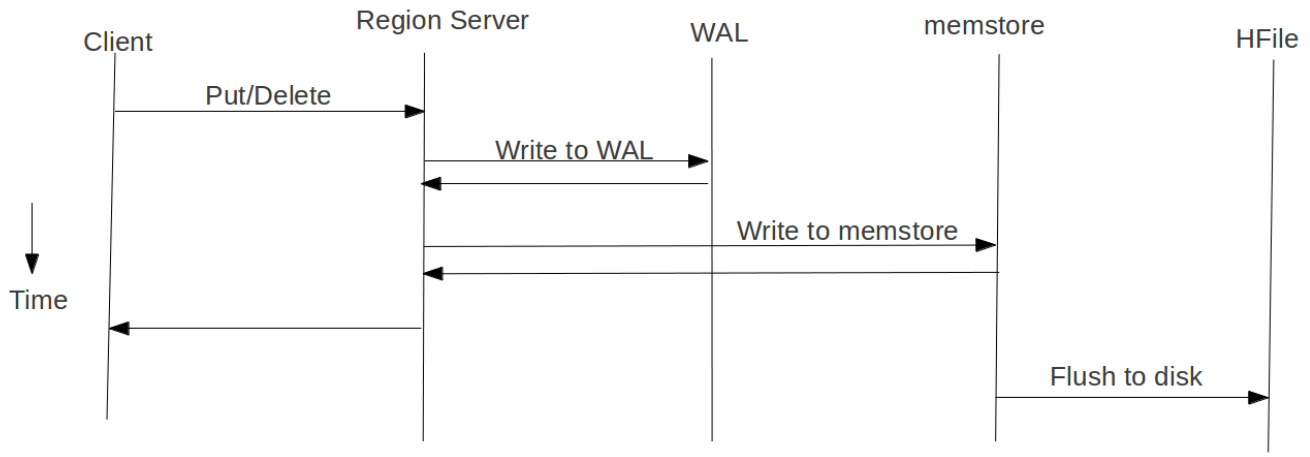
从整理上看，HBase 的数据就像是一个巨大的有序 map，所有的数据都是按照 RowKey 进行字典排序的，这些排序好的数据被进一步划分成分片或
Region。在默认情况下，当客户端发出插入或更新操作，这些请求被立即发送到对应的 RegionServer。但是为了提高吞吐量，一般我们会将数据缓存（通过关闭 autoflush 选项）在客户端，然后再以批处理的形式提交到 HBase。当 autoflush
被关闭，我们可以通过调用 flush 来提交更新请求，或者通过 hbase.client.write.buffer 参数配置缓存的大小，当缓存满了也会触发程序提交更新请求。

前面我们说到 HBase 整张表的数据都是有序的，所以我们可以通过 RowKey 来定位到对应的 RegionServer。当客户端对某个 RowKey 进行更新时，其会到 Zookeeper 里面找到 HBase 的 meta 信息表存储的 RegionServer。然后到这个 RegionServer 里面查找对应表的某个 RowKey 是由哪个 RegionServer 管理的。最后到这个 RegionServer 里面更新数据。细心的同学可能就发现，如果每次都经过这几步操作的话，那么会大大影响我们写数据的效率；而且这也会大大增加 Zookeeper 的负载。所以 HBase 客户端会缓存这些元数据，只有在元数据失效或无元数据的时候才会按照上面的流程定位到对应的 RegionServer。

当更新达到对应的 RegionServer 时，更新并不是直接写到 HFile 里面，因为 HFile 里面的数据是按照 RowKey 排序的，这有助于提升随机读的效率。而且 HFile 是存储在 HDFS 上的，并不支持直接修改。所以如果真要写到 HFile

里面，只能写到新的文件里面。但是如果每次更新都写到一个新文件，那么 HDFS 上会产生大量的小文件！所以这种方式是不支持高效扩展，并且效率极低。所以每次更新操作并不是直接写到 HFile 文件里面的。

事实上，HBase 使用一种称为 memStore 的内存结构来存储来自客户端的更新操作，因为是存储在内存，所以直接在 MemStore 里面进行随机写是非常高效的。同时，存储在 MemStore 里面的数据也是按照 RowKey 进行排序的，这个和 HFile 的组织形式一样。当 MemStore 触发了 Flush 操作的时候，存储在 MemStore 里面的数据就直接写到一个 HFile 里面，因为 MemStore 里面往往存储了很多数据，所以这个操作很高效，并能够充分展示 HDFS 的优势。需要注意的是，每个 HBase 表的每个列族对应一个 MemStore，同一个 Region 里面可能会包含多个 MemStore。



HBase Write Path

虽然按照这种方式使得我们可以快速的将数据写入到 HBase 对应的 RegionServer。但是毕竟 MemStore 是存储在内存的，所以如果对应的 RegionServer 突然挂掉了，那么没有 Flush 的数据就被丢失了，这肯定不是我们想要的。为了解决这个问题，HBase 引入了 WAL（write-ahead-log）机制，所有的更新在写入 MemStore 之前先写到 WAL 里面。而且 WAL 是直接存储在 HDFS 上的，所以通过这种机制，即使对应的 RegionServer 挂了，但是由于 WAL 的存在，数据还是可以恢复的。

在默认情况下，WAL 功能是启用的，在将 WAL 文件写入到磁盘的过程中是需要消耗一些资源的。如果数据丢失对应用程序来说不重要，那么我们是可以关掉 WAL 功能的。

WAL 文件里面的数据组织形式和 HFile 里面的是完全不一样的。WAL 文件里面包含一系列的修改，每条修改代表单个 put 或

delete。这些编辑数据包含当前修改是对应哪个 Region
的。编辑数据是按照时间编写的，因此所有的修改都是以追加的形式写到 WAL 文件的末尾。由于 WAL 里面的数据是按照修改时间编写的，所以写 WAL
文件不会发生随机写，这样可以大大提高写 WAL 的操作。

当 WAL 文件越来越大，这个文件最终是会被关闭的，然后再创建一个新的 active WAL
文件用于存储后面的更新。这个操作称为 rolling WAL 文件。一旦 WAL 文件发生了
Rolled，这个文件就不会再发生修改。

默认情况下，WAL 文件的大小达到了 HDFS 块大小的 50%（HBase 2.0.0 之前是 95%，详见 HBASE-19148），这个 WAL 文件就会发生 roll 操作。 我们可以通过
hbase.regionserver.logroll.multiplier 参数控制达到块大小的多少百分比就发生
roll。我们也可以通过 hbase.regionserver.hlog.blocksize
参数来控制块大小（注意，这个块大小不是 HDFS 的块大小）。除了文件大小能触发
rolling，HBase 也会定时去 Rolling WAL 文件，这个时间是通过
hbase.regionserver.logroll.period 参数实现的，默认是一小时。这两个策略满足一个就可以出发 WAL 的 Rolling 操作。

WAL 文件的大小对于 HBase 恢复是有影响的，因为 HBase 在使用 WAL
文件恢复数据的时候，对应的 Region 是无法提供服务的，所以尽量保持少一些的 WAL 文件。

一个 RegionServer 会包含多个 Region 的，HBase 并不为每个 Region 使用一个 WAL，而是整个 RegionServer 里面的 Regions 共用一个 WAL 日志。同时，只有一个 WAL 文件处于 active
状态。WAL 在 HDFS 上的目录格式和文件名称如下：

//WAL 目录格式

/hbase/WALs/<host>,<port>,<startcode>
/hbase/WALs/192.168.1.103,16020,1542292581331

//WAL 文件名称格式
/hbase/WALs/<host>,<port>,<startcode>/<host>%2C<port>%2C<startcode>.<timestamp>
/hbase/WALs/192.168.1.103,16020,1542292581331/192.168.1.103%2C16021%2C1547646103879.1547646379202