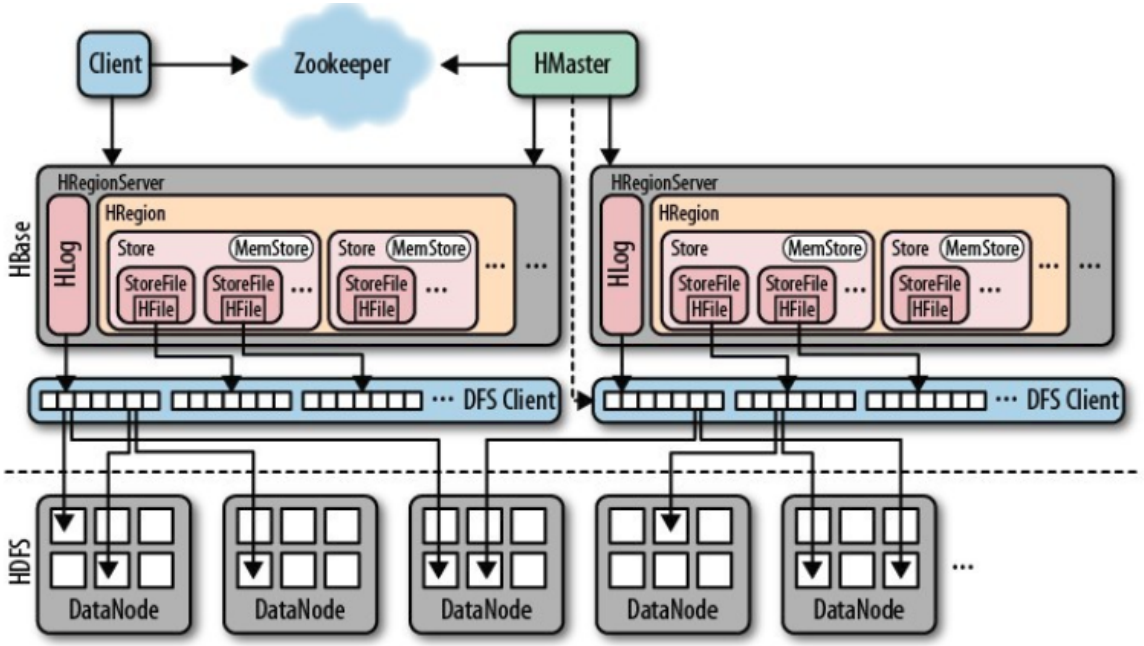
HBase 架构



1、HBase架构组件

- 从物理架构来看，HBase是包含三类服务器的主从式架构。RegionServer负责响应用户I/O请求，并向HDFS中读取数据。每当访问数据时，客户端（Client）会直接与Region Server建立链接。region的分配、DDL（创建、删除表）操作则由HMaster来处理。Zookeeper，他作为HDFS的一部分，负责维护集群的健康状态、避免HMaster单点问题
- Hadoop DataNode存储有RegionServer正在管理的数据。所有HBase的数据都存储在HDFS文件中。RegionServer和HDFS DataNode并置，这使得由RegionServer处理的数据具有数据局部性（data locality，数据被放在需要它的地方的附近）。HBase的数据在写入的时候可以从本地获取，但当它所属的region被移走时，则需要从远端获取数据，直到等到Compaction操作。
- NameNode为组成文件的物理数据块维护着他们的元数据信息

1.1 Region

- HBase表按照 行键（Row Key）被水平的分割成一个个region。每个region包含了表中从该region的开始键到结束键之间的所有行。这些region被指派给集群中的节点，这些节点便是RegionServer，它们负责数据的读写。推荐每个RegionServer管理1000个左右的region，除非它的region并不活跃。

1.2 HBase HMaster

- Region的分配、DDL（创建、删除表）操作由HBase Master处理。每个master的职责都包括有：
  - ①、协调RegionServer
    - 在启动或需重分配region的时候负责region的分配，以实现数据恢复和负载平衡
    - 监视所有在集群中的RegionServer（监听来自Zookeeper的消息）
  - ②、管理功能

- 管理用户对表的增删改查操作

1.3 Zookeeper：协调者

- HBase用Zookeeper来提供分布式协调服务，以维护集群中服务器的状态。Zookeeper留存了服务器健康状态与是否可用的消息，并提供服务器故障通知。Zookeeper用consensus协议来保障共享状态。需要注意的是，一份consensus协议需要3个或5个机器参与

2、各组件如何协调工作

- Zookeeper用来协调分布式系统成员的共享状态信息。RegionServer和处于活跃状态的HMaster通过一个会话（session）与Zookeeper建立连接。Zookeeper通过heartbeat来为活跃的会话维护短暂的临时节点。

3、HBase的首次读/写

- META表 是一个特殊的HBase编目表，它保存着集群中region的地址。META表的地址会保存在Zookeeper中。当客户端首次在HBase中读/写时，会执行如下操作：
  - ①、客户端获取Zookeeper中的META表所记录的RegionServer
  - ②、客户端查询METAServer，以找到它想要的行键位置上的RegionServer。他将会连同META表的位置一起缓存（cache）这些信息
  - ③、客户端从相应的RegionServer得到相应的行键。

  在以后的读取中，客户端就可以通过cache重新获得META的地址以及之前读到的行键信息。除非有由region移动所引起的miss，那么随着时间的推移，客户端就不必再去查询META表了；取而代之的是他会反复查询并不断更新cache。

4、HBase Meta表

- META表是一个特殊的HBase表，它维护着记录系统中的所有region的一个列表
- META表的数据结构类似于B树
- META表的架构组成如下
  - 键：region的开始键，region ID（table，key，region）
  - 值：RegionServer，region server

5、RegionServer组件

- 每个RegionServer都运行在HDFS的一个数据节点上，RegionServer由如下部分构成：
  - ①、WAL：分布式文件系统上的文件，WAL保存着一些临时的新数据，以用作之后故障的修复。
  - ②、BlockCache：BlockCache是读操作是的cache，它保存着内存中经常被读的数据。当cache填满后，最近最少使用的数据将会被剔除。
  - ③、MemStore：写操作时的cache。他保存着尚未写入磁盘的新数据，这些数据在被写入磁盘前是有序的。每个region的每一列族（column family）都有一个MemStore
  - ④、Hfiles：保存着磁盘上有序键值对的行信息

6、HBase写操作

- 当客户端发生Put操作请求是，首先数据会被写入WAL：

    - 更改被添加在磁盘上WAL文件的末端
    - WAL恢复服务器崩溃事务中的 为持久化（not-yet-persisted）的数据
- 一旦数据被写入WAL，他们就会被放在MemStore中。接着，Put请求的确认信息（ACK）将会回复给客户端

7、HBase MemStore

- MemStore键内存中的更新（updates）按照有序键值对保存，这和它在HFile中的存储是一样的模式。每一列族有一个MemStore，更新按照每一列组有序存储。

8、HBase Region Flush

- 当MemStore积攒了足够多的数据，所有的有序集会被写入HDFS。HBase的每一列族中有多个HFile，里面包含由一定数据量的单元数据（cells），或说键值对。当MemStores中的有序键值对被flush到磁盘上时，就生成了这些HFile文件。
- 这便是HBase的列族有数量限制的原因之一。每个列族有一个MemStore被填满后，它们会全部flush到磁盘上。MemStroe仍保存有最后写入的序列号，因而系统能够知道哪些数据依旧是持久的（写到磁盘上的）
- 最高序列号在每个HFile中被存为一个元字段（meta field），这些字段反映出何处持久性终止了以及应该在何处继续。当一个region启动的时候，它的序列号被读取，而它的最高序列号则被作为新更改（edits）时的序列号。

9、HBase HFile

- 数据被保存在HFile中，它们包含有序键/值。当MemStore积攒到足够多的数据，所有的有序集会写入HDFS的一个新HFile中。写入是顺序写入；由于它避免了磁盘驱动器磁头的移动，所以写入的速度非常快

9.1 HBase Region Flush

- 每个HFile包含有一个多层索引（index），这使得HBase不用去读完全部的文件就能找到想要的数据。多层索引的结构类似于一个B+树
    - 键值对被按照升序存储
    - 索引通过行键指向64KB-blocks（块）中的键值数据；
    - 每一个block都有它自己的叶子索引；
    - 每个block的最后一个键被放在中间索引（intermediate index）中
    - 根索引指向中间索引
- HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。Trailer指向元块（meta blocks），它被写在文件的最后。Trailer还有着类似于bloom filter的信息以及时间段信息（time range info）。Bloom filter可用于跳过那些没有确定键值的文件。时间段信息可用于跳过那些不在所需时间段内的文件。

9.2 HFile索引

- HFile打开和保存至内存中时加载进来。这使得查找能在单个磁盘上进行。

10、HBase读合并（Read Merge）

- 每行对应的键值对单元能被放置在多出。行单元被保存在HFile中；最近更新的行单元被放在MemStore中；最近读取单元被放在BlockCache中。那么但需要读取一行时，系统是如何找到相应的单元并返回的呢？读操作将来自BlockCache、MemStore和HFiles的键值对合并，步骤如下：

    - ①、扫描器从BlockCache中找到行单元，最近读取的键值对就被缓存在此。当内存需要时，最近最少使用的数据就会被剔除。
    - ②、扫描器会检查MemStore，MemStore中包含有大多数最近写入的数据。
    - ③、如果扫描器没能在BlockCache和MemStore中找全所有的行单元，HBase就会用BlockCache索引和bloom filters把HFiles加载进内存，因为这些HFiles中可能包含所需的行单元。
- 每个MemStore中有多个HFiles，这意味着在读操作时，需要检查多个文件，这将会影响时间性能。这被称为读出放大。

11、HBase Compaction

11.1 Minor Compaction

- HBase会自动拾取一些较小的HFile，并将它们重新写入一些较大的HFiles中。这个过程被称为minor compaction。Minor compaction通过重写操作，利用合并排序将较小的文件转化为较大但数量较小的文件中。

11.2 Major Compaction

- Major Compaction将一个region里的全部HFiles，按照每一列族合并和从重写成一个HFiles。在这个过程中，会将已经删除或者过期的单元剔除。这样优化了读操作的性能；然而，由于major compaction重写了所有的文件，因而在这过程中可能会出现较大的磁盘I/O和网络拥堵。这被称之为 写入放大（Write amplification）
- Write amplification可被设置为自动运行。由于写入放大，major compactions往往被安排在周末或者晚上进行。Major compaction能够使得那些由于服务器故障或者负载均衡而较远的数据，变得相对RegionServer处于本地。

12、Region拆分（split）

- 起初，每个表只有一个region。当一个region逐渐变得过大时，他会分裂成两个子region。这两个子region分别代表了原region的二分之一，它们在RegionServer上被并行的打开，之后这个拆分会告知HMaster。
- 由于负载均衡的因素，HMaster可能会将新的region移除到其他服务器。这导致新的在RegionServer需要从远处的HDFS节点中获取数据，除非major compaction将这些数据移动到RegionServer的本地节点。

13、HDFS数据备份

- 所有独活写操作的对象都是主节点（primary node）。HDFS备份了WAL和HFile块，HFile块的备份时自动进行的。HBase通过以来HDFS来存储文件时保障数据安全。当数据被写入HDFS时，一份数据拷贝被写在本地，接着他会备份到二级节点，第三份拷贝会写入其他节点。

14、HBase崩溃修复

- 当一个RegionServer故障时，再采取探测和修复操作前，崩坏的region不可用。Zookeeper会在它失去RegionServer的heartbeats时，确定故障节点。接着HMaster将会被告知RegionServer已经故障。
- 当HMaster检测到某个RegionServer出现故障，HMaster会重新分配从故障服务器到活跃服务器的region。为了修复RegionServer的MemStore中还没保存到磁盘上的更改，HMaster会把故障服务器的WAL分成单独的文件，并将这些文件保存至新的RegionServer数据节点上。每个RegionServer从分割出的WAL中回收WAL，来为损坏的region重建MemStore

15、数据修复

- WAL文件中包含有一个更改列表，每个更改（edit）表示一个单独的put或者delete操作。更改按照时间顺序记录，为了保持持久性，新增加的操作被放在磁盘上WAL文件的尾端
- 如果当数据在内存中去不在HFile中时出现了故障时，WAL文件会回放，回放WAL通过读WAL，添加并排序当前MemStore中的更改实现。最后。MemStore会将写入更改flush进HFile。

16、HBase架构评测

- 优势：
  - ①、强一致模型
    - 当写操作返回时，所有的读者都能看到相同的值
  - ②、自动化规模
    - 当数据量过大时，会自动分割Region
    - 利用HDFS来存储和备份数据
  - ③、内置恢复
    - WAL
  - ④、由Hadoop集成
    - 在HBase上运用MapReduce是容易的
- 不足：
  - WAL回放较慢
  - 复杂的故障修复较慢
  - Major Compaction的I/O放大

17、 HBase读写流程

- HBase使用MemStore和StoreFile存储对表的更新，数据在更新时首先写入HLog和MemStore。MemStore中的数据时排序的，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队列中，有单独的线程Flush到磁盘上，成为一个StoreFile。与此同时，系统会在Zookeeper中记录一个CheckPoint，表示这个时刻之前的数据变更已经持久化了。当系统出现意外时，可能导致MemStore中的数据丢失，此时使用HLog来恢复CheckPoint之后的数据。
- StoreFile是只读的。一旦创建后就不可以在修改了。因此HBase的更新其实是不断追加的操作。当一个Store中StoreFile达到一定与之后，就会进行一次合并操作，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进行切分操作，等分为两个StoreFile。

17.1 写操作流程

- ①、Client通过Zookeeper的调度，向RegionServer发出写数据请求，在Region中写数据。
- ②、数据被写入Region的MemStore，直到MemStore达到预设阈值。
- ③、MemStore中的数据被Flush成一个StoreFile
- ④、随着StoreFile文件的不断增加，档期数据量增加到一定阈值后，出发Compact合并操作，将多个StoreFile合并成一个StoreFile，同时进行版本的合并和数据删除。
- ⑤、StoreFiles通过不断Compact合并操作，逐步形成越来越大的StoreFile。
- ⑥、单个StoreFile大小超过一定阈值后，出发Split操作，把当前Region Split成两个新的Region。父Region下线，新Split出的两个子Region会被HMaster分配到相应的RegionServer上，使得原先1个Region的压力得以分流到两个Region上。

- 可以看出HBase只有增添数据，所有的更新和删除操作都是在后续的Compact历程中举行的，使得用户的写操作只要进入内存就可以立刻返回，实现了HBase I/O的高机能。

17.2 读操作流程

- ①、Client访问Zookeeper，查找-ROOT-表，获取.META表信息。
- ②、从META表查找，获取存放目标数据的Region信息，从而找到对应的RegionServer。
- ③、通过RegionServer获取需要查找的数据。
- ④、RegionServer的内存分为MemStore和BlockCache两部分。MemStore主要用于写数据，BlockCache主要用于读数据。读请求先到MemStore中查数据，查不到就到BlockCache中查，再查不到就回到StoreFile上读，并把读的结果放入BlockCache。