大讲台
dajiangtai.com

HBase原理和架构

主讲人：杨老师

01 HBase概述

为什么使用NoSQL数据库？

- RDBMS的缺点

➢ 高并发读写的瓶颈

RDBMS应付每秒上万次的SQL查询，勉强可以支撑的住；但是每秒应付上万次的SQL写数据操作，硬盘I/O难以承受。

➢ 扩展性的瓶颈

目前互联网网站都是24小时不间断的提供服务，对RDBMS很难进行升级和扩展，因为这往往涉及到停机维护和数据迁移。

➢ 事务的副作用

在RDBMS中，事务可以确保数据的完整性和一致性，但是也会造成性能的急剧下降，因为它需要高额的管理成本。

为什么使用NoSQL数据库？

NoSQL数据库的优势

- 高并发读写

NoSQL数据库具有非常良好的读写性能，尤其在海量数据。主要是由于它是弱关系性，数据库的结构简单。NoSQL的Cache是记录级的，是一种细粒度的Cache，数据可以直接写入缓存，速度比较快，Cache不失效。而MySQL使用Query Cache，是一种粗粒度的Cache，当表进行更新操作的时候，Cache就会失效，在频繁的交互应用中，Cache性能不高。

- 扩展性强

NoSQL数据库有很多种，但数据都没有很强的关系特性，数据之间是弱关系，非常容易扩展。比如HBase、Cassandra扩展都非常容易。

- 数据模型比较灵活

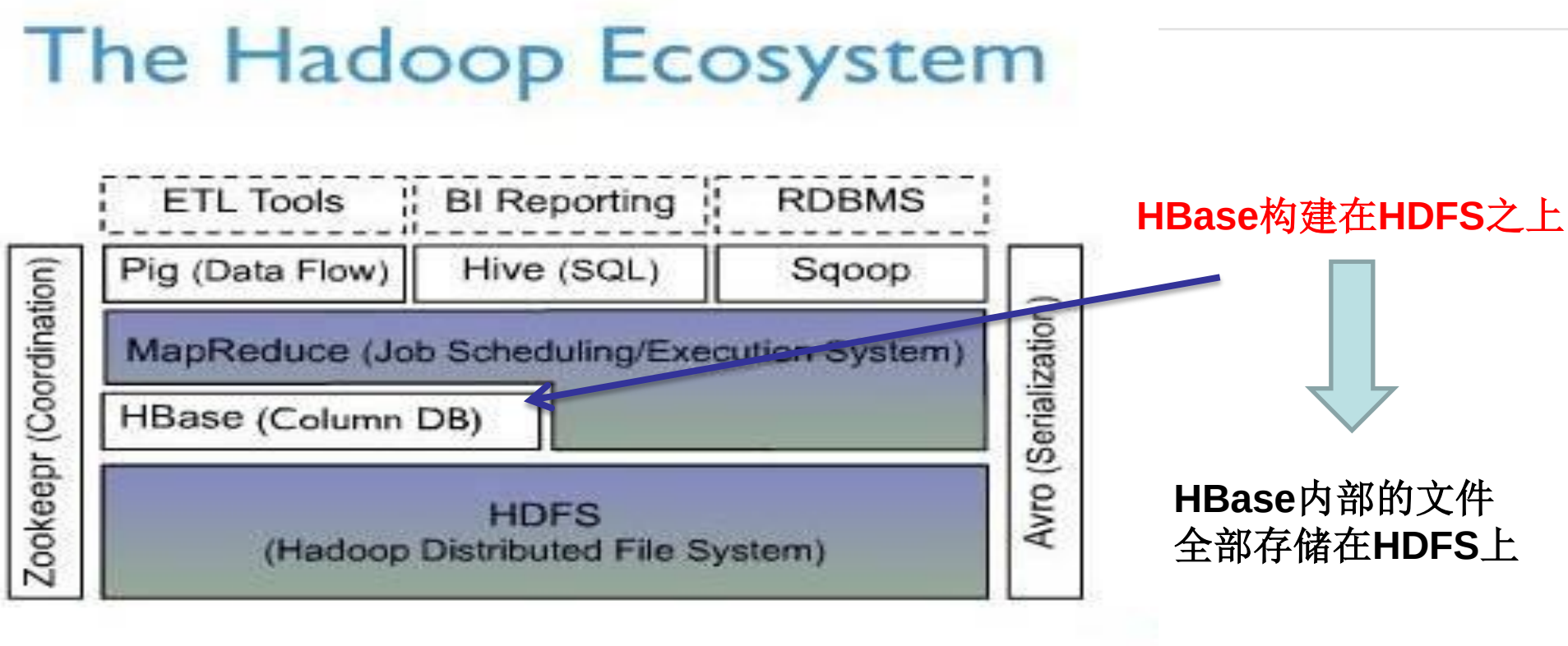
NoSQL数据库不需要预先创建字段，可以存储时再自定义数据格式。在RDBMS中，增删字段非常麻烦，特别是海量数据的情况下。

HBase是什么

- HBase是构建在HDFS之上的分布式列存储数据库，是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，利用HBase技术可以在廉价PC Server上搭建起大规模结构化存储集群。

- HBase 是Google Bigtable的开源实现，类似Google Bigtable利用GFS作为其文件存储系统，Google运行MapReduce来处理Bigtable中的海量数据，HBase同样利用Hadoop MapReduce来处理HBase中的海量数据；Google Bigtable利用 Chubby作为协同服务，HBase利用Zookeeper作为对应。

HBase在生态体系中的位置



•上图描述了Hadoop EcoSystem中的各层系统，其中HBase位于结构化存储层，Hadoop HDFS为HBase提供了高可靠性的底层存储支持，Hadoop MapReduce为HBase提供了高性能的计算能力，Zookeeper为HBase提供了稳定服务和failover（故障切换）机制。

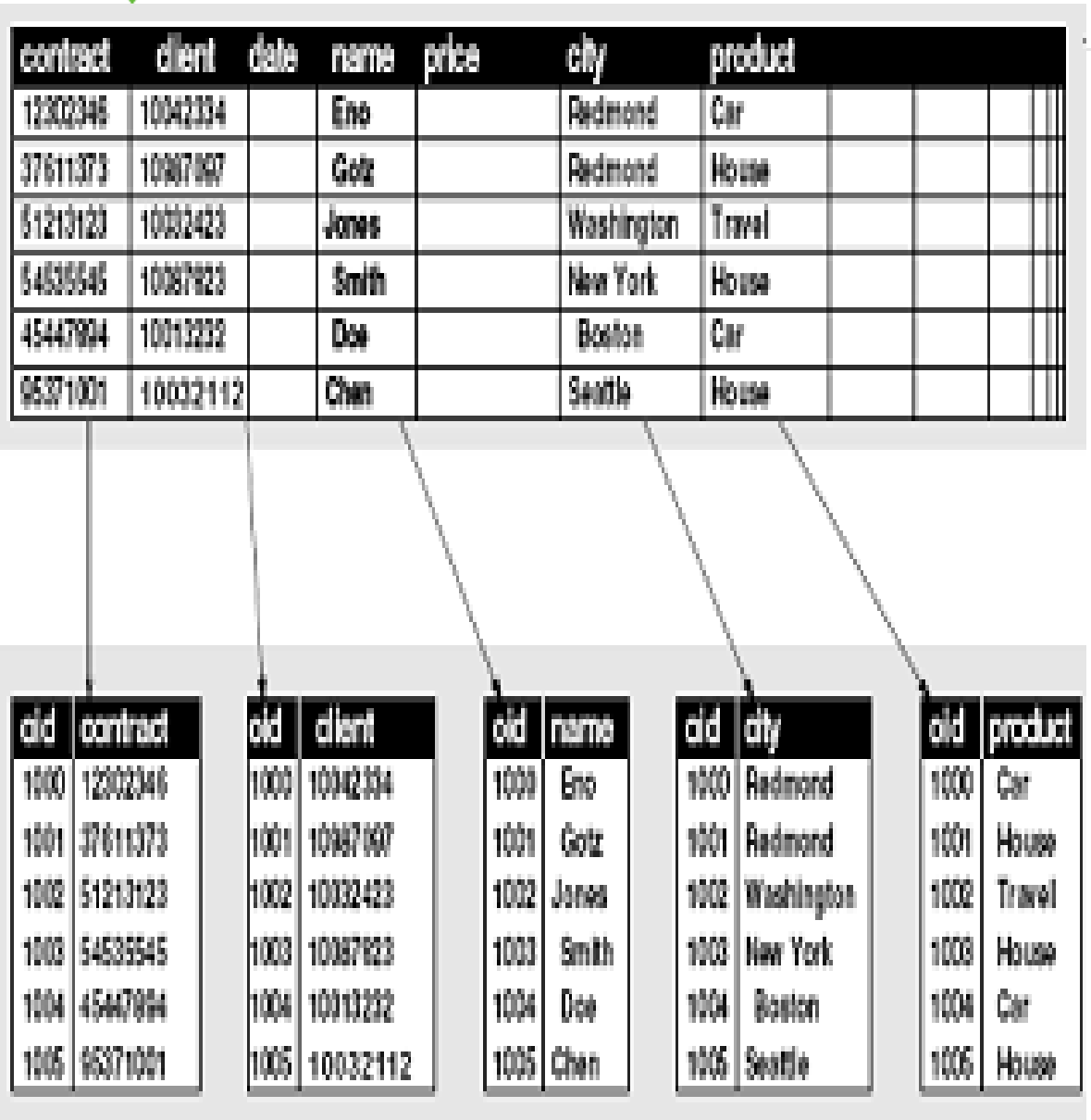
•此外，Pig和Hive还为HBase提供了高层语言支持，使得在HBase上进行数据统计处理变的非常简单。

•Sqoop则为HBase提供了方便的RDBMS数据导入功能，使得传统数据库数据向HBase中迁移变的非常方便

HBase表的特点

- 大：单表可以数十亿行，数百万列
- 无模式：同一个表的不同行可以有截然不同的列
- 面向列：存储、权限控制、检索均面向列
- 稀疏：空列不占用存储，表是稀疏的
- 多版本：每个单元中的数据可以有多个版本，默认情况下版本号自动分配，是单元格插入时的时间戳
- 数据类型单一：数据都是字符串，没有类型

行存储和列存储



- 数据是按行存储的
- 没有索引的查询，使用大量I/O
- 建立索引和物化视图需要花费大量时间和资源
- 面向查询的需求，数据库必须被大量膨胀才能满足性能要求

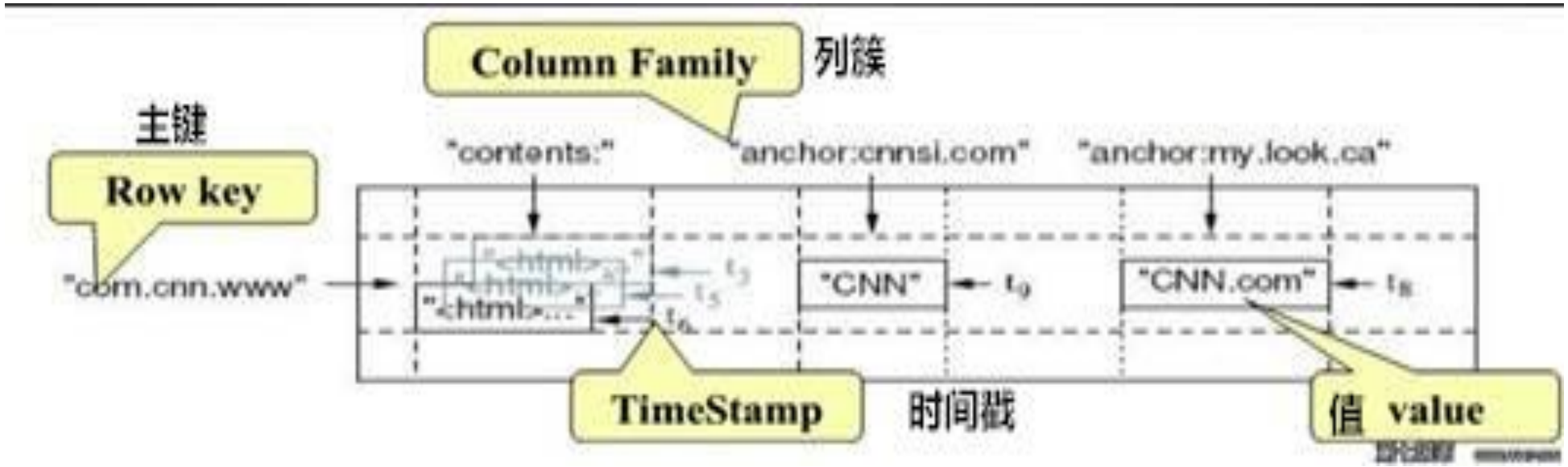
- 数据是按列存储-每一列单独存放
- 数据即是索引
- 只访问查询涉及的列-大量降低系统I/O
- 每一列由一个线索来处理-查询的并发处理
- 数据类型一致，数据特征相似-高效压缩

02 HBase逻辑模型

HBase逻辑模型

HBase中最基本的单位是列，一列或者多列构成了行，行有行键（Rowkey），每一行的行键都是唯一的，对相同行键的插入操作被认为是对同一行的操作，多次插入操作其实就是对该行数据的更新操作。

HBase中的一个表有若干行，每行有很多列，列中的值可以有多个版本，每个版本的值称为一个单元格，每个单元格存储的是不同时间该列的值。



Hbase表包含两个列簇：contents和anchor。在该示例中，列簇anchor有两个列（anchor:cnnsi.com和anchor:my.look.ca），列簇contents仅有一个列contents：html。其中，列名是由列簇前缀和修饰符（Qualifier）连接而成，分隔符是英文冒号。例如，列anchor:my.look.ca是列簇anchor前缀和修饰符my.look.ca组成。所以在提到HBase的列的时候应该使用的方式是”列簇前缀+修饰符”。

03 HBase数据模型核心概念

HBase数据模型的核心概念

Row Key	version	Column Family	ColumnFamily
		URI	Parser
r1	t2	url=http://www.taobao.com	title=天天特价
	t1	host=taobao.com	
r2	t5	url=http://www.alibaba.com	content=每天…
	t4	host=alibaba.com	

- 表（table）

在HBase中数据是以表的形式存储的，通过表可以将某些列放在一起访问，同一个表中的数据通常是相关的，可以通过列簇进一步把列放在一起进行访问，。用户可以通过命令行或者Java API来创建表，创建表时只需要指定表名和至少一个列簇。

HBase的列式存储结构允许用户存储海量的数据到相同的表中，而在传统数据库中，海量数据需要被切分成多个表进行存储。

HBase数据模型的核心概念

- 行健（Row Key）

Rowkey既是HBase表的行键，也是HBase表的主键。HBase表中的记录是按照Rowkey的字典顺序进行存储的。

在HBase中，为了高效地检索数据，需要设计良好的Rowkey来提高查询性能。首先Rowkey被冗余存储，所以长度不宜过长，Rowkey过长将会占用大量的存储空间同时会降低检索效率；其次Rowkey应该尽量均匀分布，避免产生热点问题；另外需要保证Rowkey的唯一性。

- 列簇（ColumnFamily）

HBase表中的每个列都归属于某个列簇，一个列簇中的所有列成员有着相同的前缀。

比如，列url和host都是列簇URI的成员。列簇是表的schema的一部分，必须在使用表之前定义列簇，但列却不是必须的，写数据的时候可以动态加入。一般将经常一起查询的列放在一个列簇中，合理划分列簇将减少查询时加载到缓存的数据，提高查询效率，但也不能有太多的列簇，因为跨列簇访问是非常低效的。

HBase数据模型的核心概念

- 单元格

HBase 中通过 Row 和 Column 确定的一个存储单元称为单元格（Cell）。每个单元格都保存着同一份数据的多个版本，不同时间版本的数据按照时间顺序倒序排序，最新时间的数据排在最前面，时间戳是64位的整数，可以由客户端再写入数据时赋值，也可以由RegionServer自动赋值。

为了避免数据存在过多版本造成的管理（包括存储和索引）负担，HBase 提供了两种数据版本回收方式。 一是保存数据的最后 n 个版本；二是保存最近一段时间内的数据版本，比如最近七天。用户可以针对每个列族进行设置。

04 HBase数据模型操作

HBase支持的操作

- 所有操作均是基于rowkey的；
- 支持CRUD（Create、Read、Update和Delete）和Scan；
- 单行操作
  - ✓ Put
  - ✓ Get
  - ✓ Scan—含头不含尾
- 多行操作
  - ✓ Scan
  - ✓ MultiPut
- 没有内置join操作，可使用MapReduce解决。

05 HBase物理模型

HBase物理模型

- 每个column family存储在HDFS上的一个单独的文件里
- Rowkey和version在每个column family里均有一份
- 空值不保存，占位符都没有

HBase 为每个值维护了多级索引，即：
<key, column family, column name, timestamp>

Table 5.3. ColumnFamily contents

Row Key	Time Stamp	ColumnFamily "contents:"
"com.cnn.www"	t6	contents:html = "<html>..."
"com.cnn.www"	t5	contents:html = "<html>..."
"com.cnn.www"	t3	contents:html = "<html>..."

Table 5.2. ColumnFamily anchor

Row Key	Time Stamp	Column Family anchor
"com.cnn.www"	t9	anchor:cnnsi.com = "CNN"
"com.cnn.www"	t8	anchor:my.look.ca = "CNN.com"

例子

Row key	Data
cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
tlipcon	info: { 'height': '5ft7', 'state': 'CA' } roles: { 'Hadoop': 'Committer'@ts=2010, 'Hadoop': 'PMC'@ts=2011, 'Hive': 'Contributor' }

`info` Column Family

Row key	Column key	Timestamp	Cell value
cutting	info:height	1273516197868	9ft
cutting	info:state	1043871824184	CA
tlipcon	info:height	1273878447049	5ft7
tlipcon	info:state	1273616297446	CA

`roles` Column Family

Row key	Column key	Timestamp	Cell value
cutting	roles:ASF	1273871823022	Director
cutting	roles:Hadoop	1183746289103	Founder
tlipcon	roles:Hadoop	1300062064923	PMC
tlipcon	roles:Hadoop	1293388212294	Committer
tlipcon	roles:Hive	1273616297446	Contributor

Sorted on disk by Row key, Col key, descending timestamp

物理存储

- Table中的所有行都按照row key的字典序排列；
- Table 在行的方向上分割为多个Region；

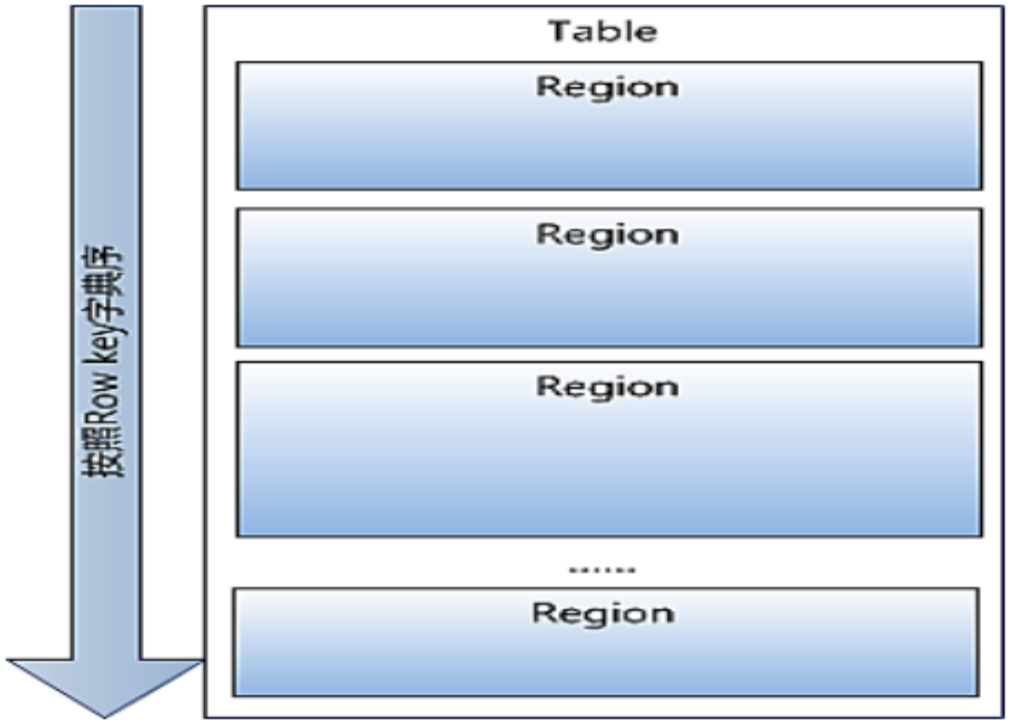
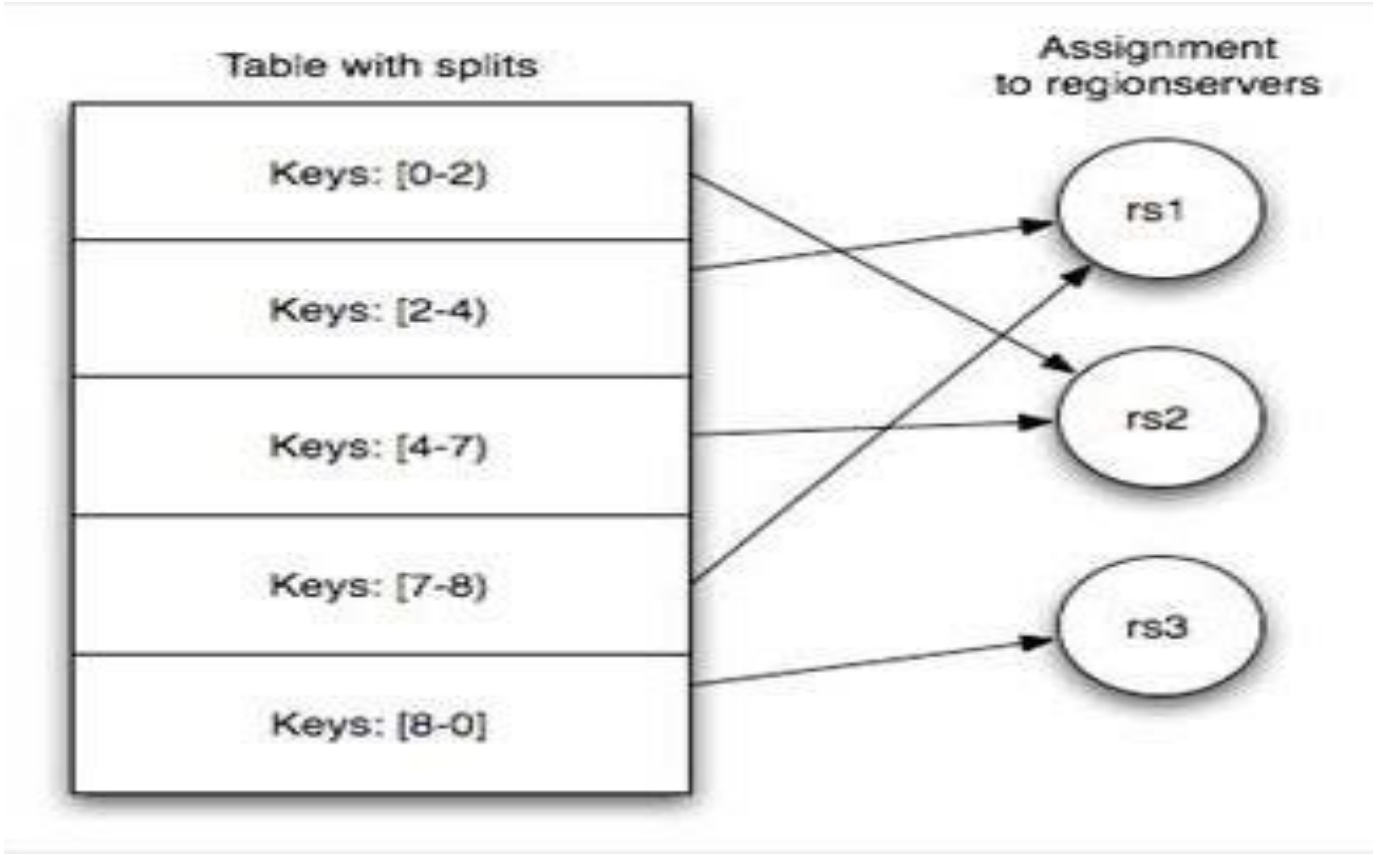


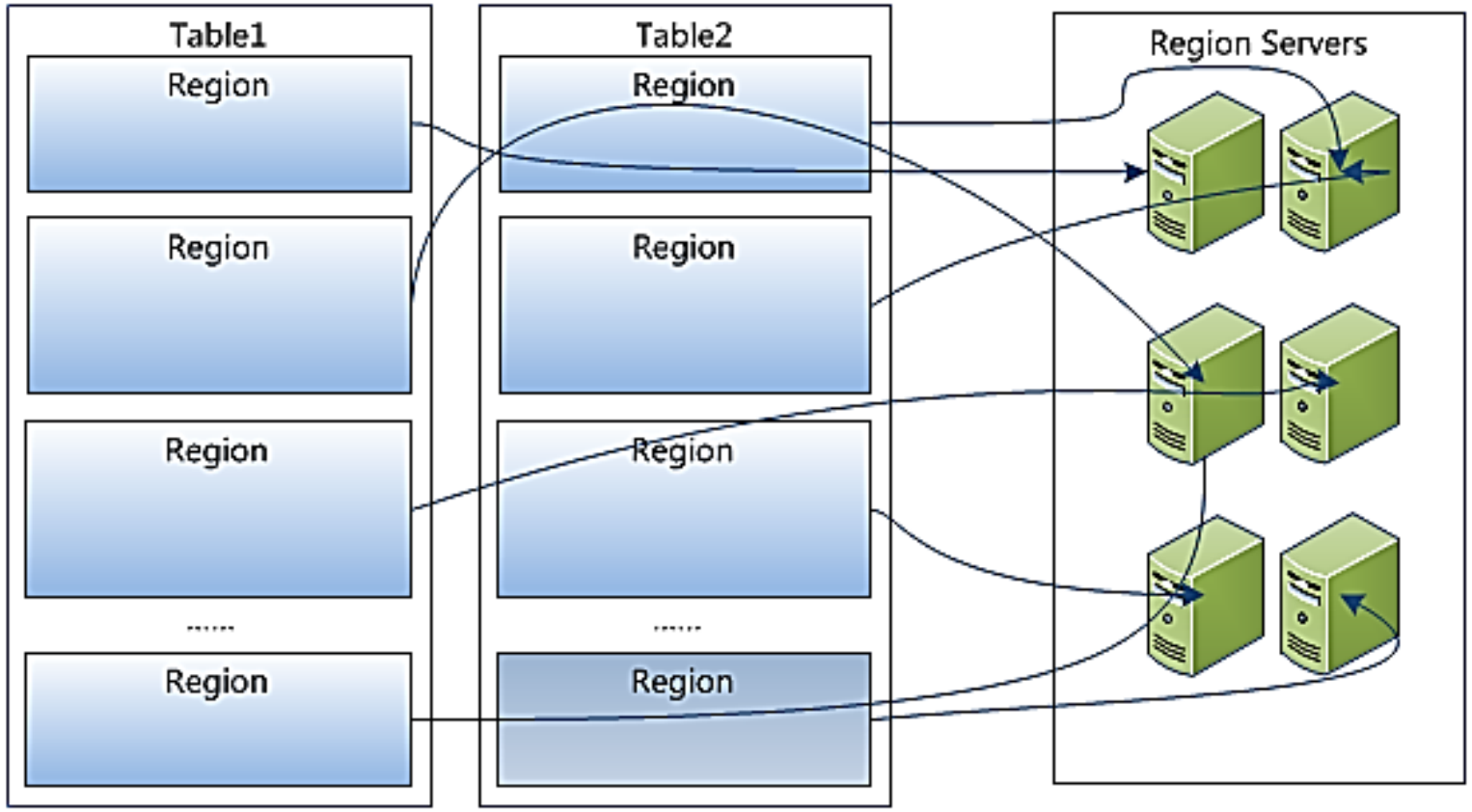
Table vs Region

- Table默认最初只有一个Region，随着记录数不断增加而变大后，会逐渐分裂成多个region，一个region由[startkey,endkey]表示，不同的region会被Master分配给相应的RegionServer进行管理



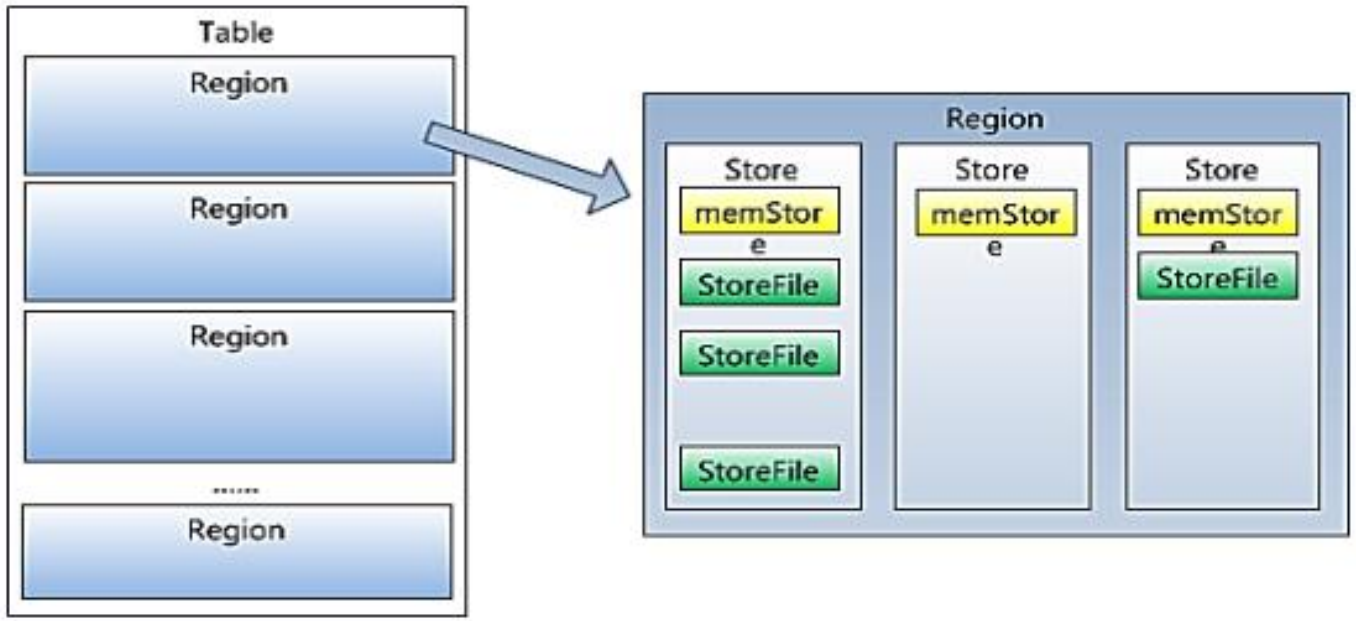
物理存储

- Region是HBase中分布式存储和负载均衡的最小单元。不同Region分布到不同RegionServer上



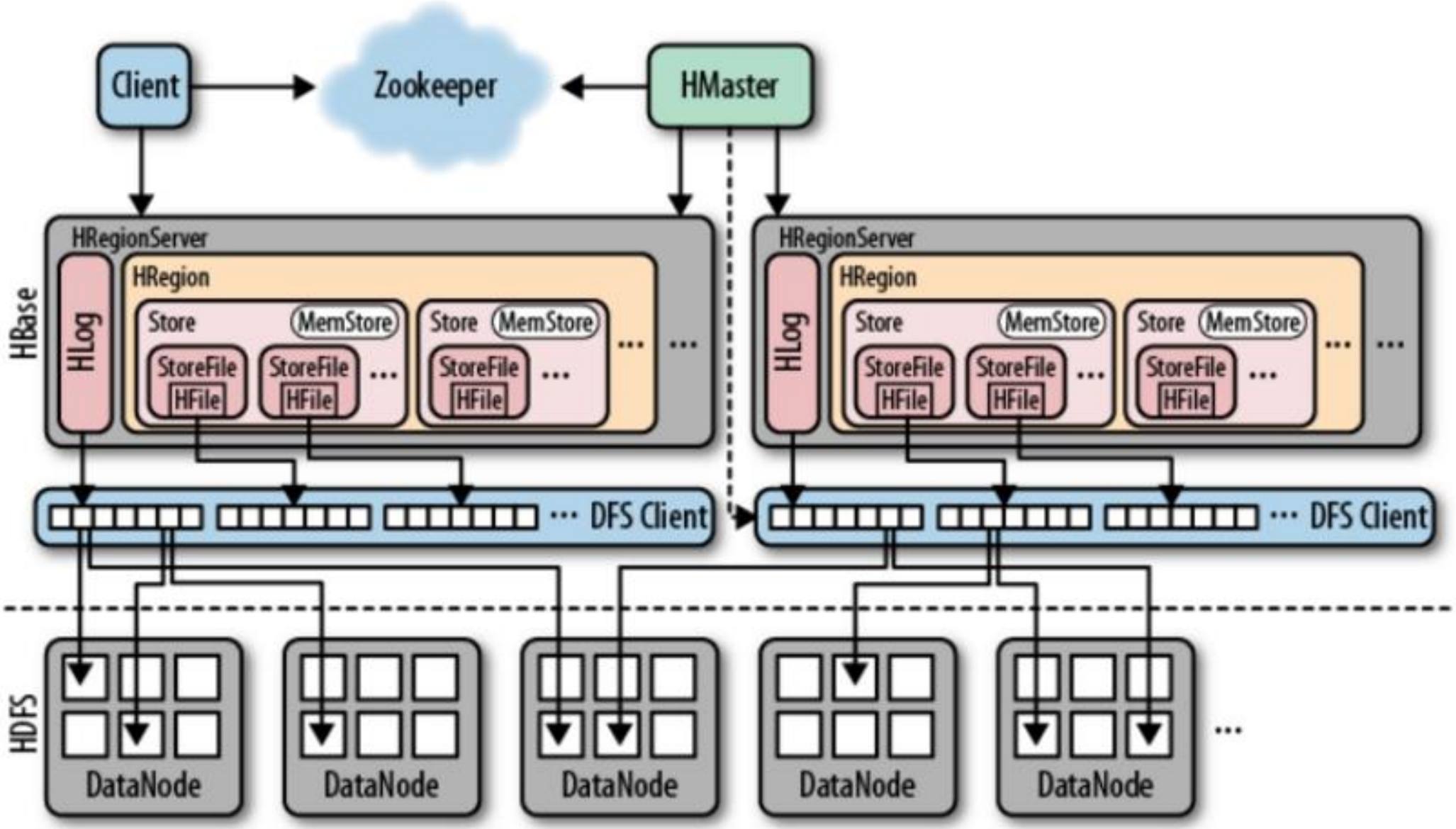
物理存储

- Region虽然是分布式存储的最小单元，但并不是存储的最小单元。
- Region由一个或者多个Store组成，每个store保存一个columns family
- 每个Strore又由一个memStore和0至多个StoreFile组成；
- memStore存储在内存中，StoreFile存储在HDFS上。



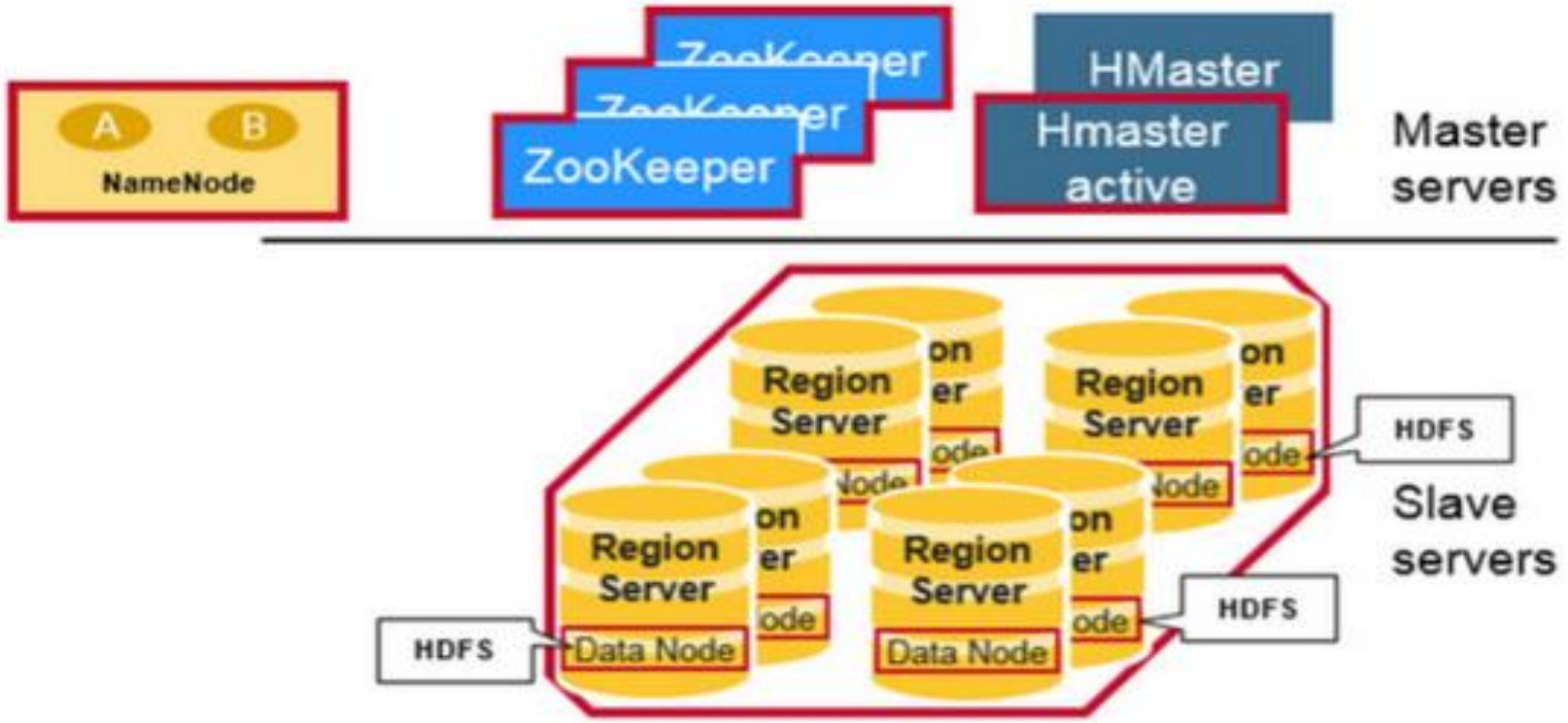
06 HBase系统架构

HBase 系统架构图



HBase 系统架构

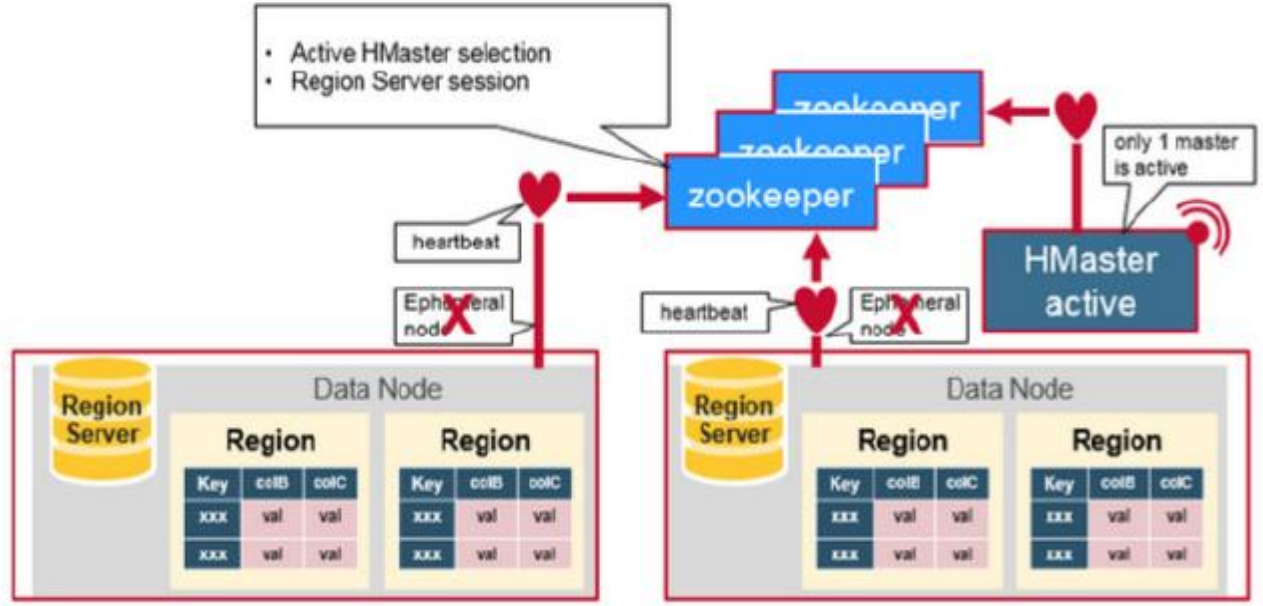
HBase采用Master/Slave架构搭建集群，由HMaster节点、HRegionServer节点、ZooKeeper集群组成，而在底层它将数据存储于HDFS中，因而涉及到HDFS的NameNode、DataNode等，每个DataNode上面最好启动一个HRegionServer, 这样在一定程度上保持数据的本地性。



HBase 系统架构—Zookeeper

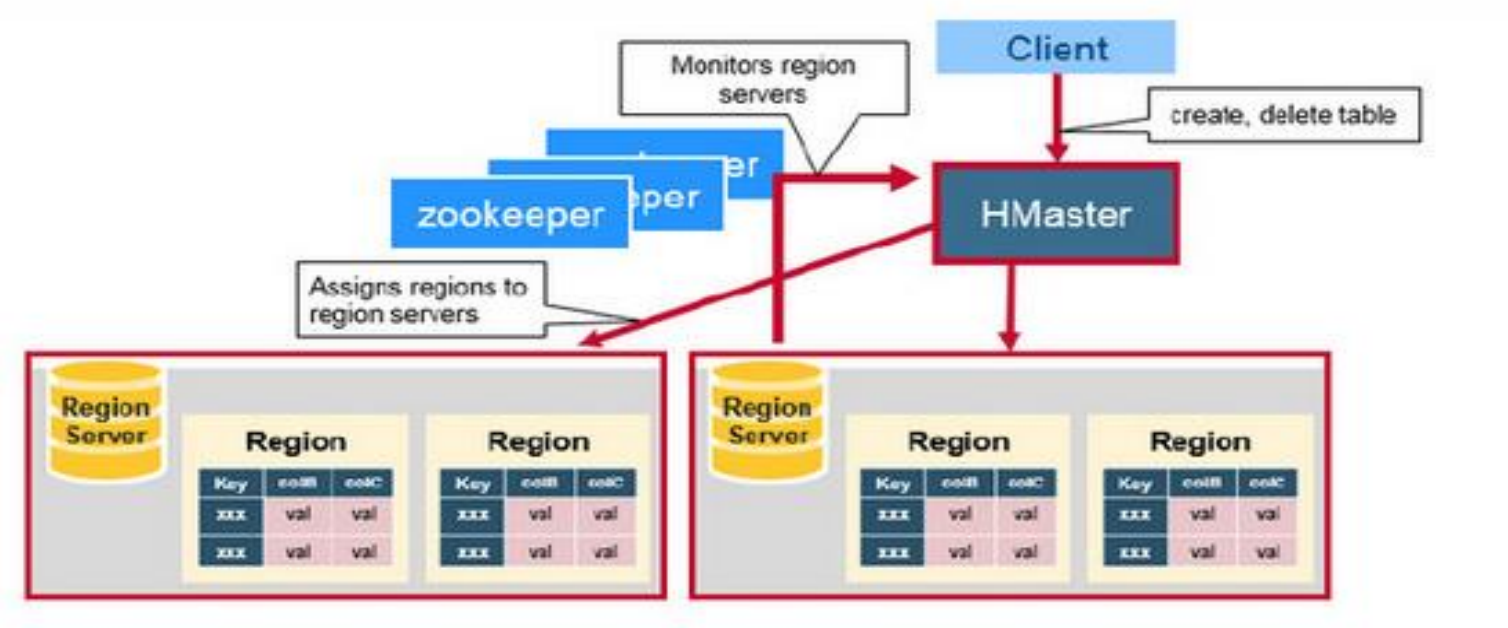
ZooKeeper协调集群所有节点的共享信息，在HMaster和HRegionServer连接到ZooKeeper后创建Ephemeral节点，并使用Heartbeat机制维持这个节点的存活状态，如果某个Ephemeral节点实效，则HMaster会收到通知，并做相应的处理。

HMaster通过监听ZooKeeper中的Ephemeral节点(默认：/hbase/rs/*)来监控HRegionServer的加入和宕机。在第一个HMaster连接到ZooKeeper时会创建Ephemeral节点(默认：/hbasae/master)来表示Active的HMaster，其后加进来的HMaster则监听该Ephemeral节点，如果当前Active的HMaster宕机，则该节点消失，因而其他HMaster得到通知，而将自身转换成Active的HMaster，在变为Active的HMaster之前，它会创建在/hbase/back-masters/下创建自己的Ephemeral节点。



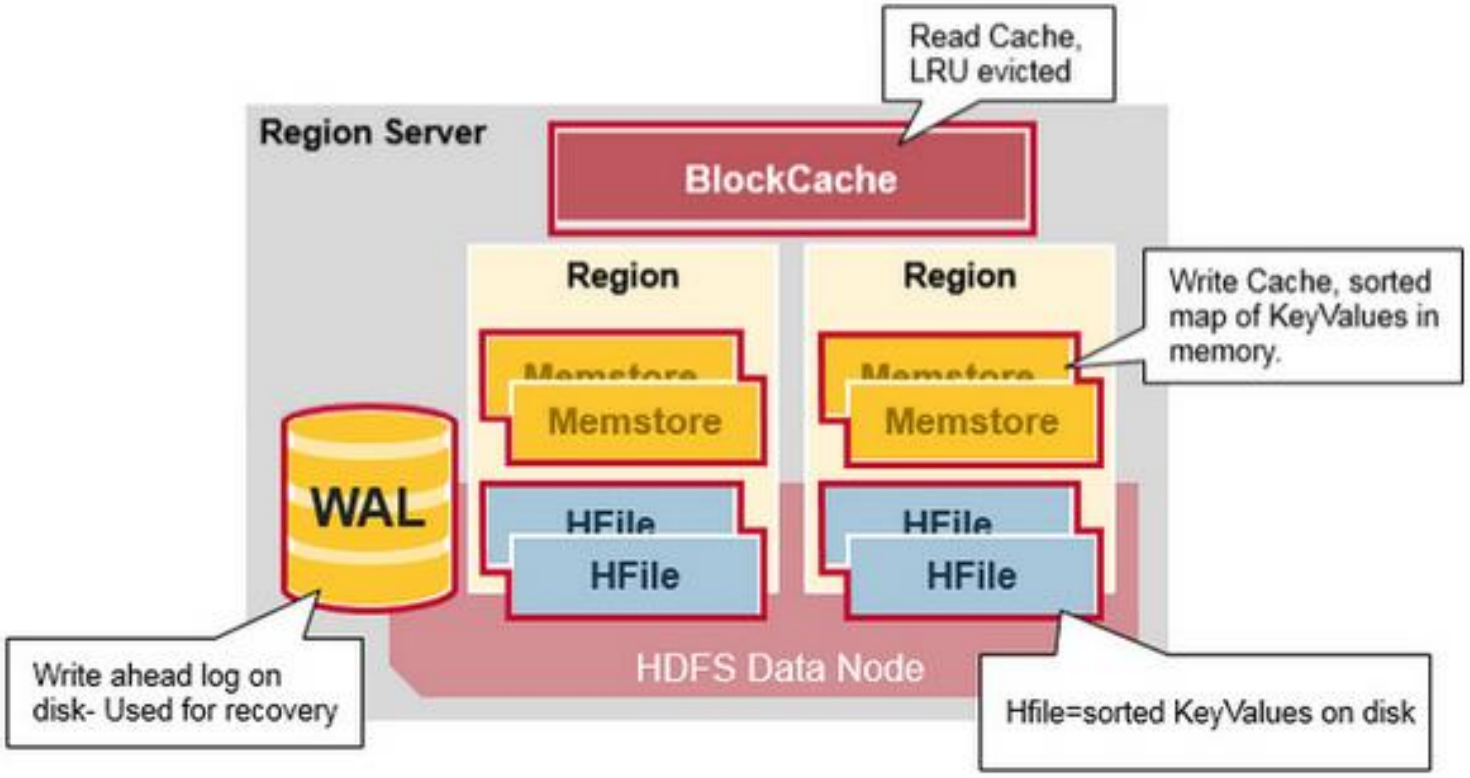
HBase 系统架构—Master

- 管理HRegionServer，实现其负载均衡。
- 管理和分配HRegion，在HRegion split时分配新的HRegion；在HRegionServer退出时迁移其内的HRegion到其他HRegionServer上。
- 监控集群中所有HRegionServer的状态(通过Heartbeat和监听ZooKeeper中的状态)
- 实现DDL操作（Data Definition Language，namespace和table的增删改，columnfamiliy的增删改等）。
- 管理namespace和table的元数据（实际存储在HDFS上）。权限控制（ACL）



HBase 系统架构—HRegionServer

- Region server维护Master分配给它的region，处理对这些region的IO请求
- Region server负责切分在运行过程中变得过大的region
- HRegionServer一般和DN在同一台机器上运行，实现数据的本地性
- HRegionServer包含多个HRegion，由WAL(HLog)、BlockCache、MemStore、HFile组成



RegionServer组成

- WAL：Write Ahead Log即提前写日志（Log），根据字面意思就知道，在写操作的时候，就是先要写入到该日志文件中。所有写操作都会先保证将数据写入这个Log文件后，才会真正更新MemStore，最后写入HFile中。这样可以在RegionServer挂掉后，通过WAL来恢复数据，从而避免数据的丢失。一般一个RegionServer只有一个WAL实例，也就是说一个RegionServer的所有WAL写都是串行的，你可能会觉得这会有性能问题，因而在HBase1.0之后，通过HBASE-5699（ http://hbase.apache.org/book.html#hbase.versioning）实现了多个WAL并行写(MultiWAL)，该实现采用HDFS的多个管道写，以单个HRegion为单位。Log文件会定期Roll出新的文件而删除旧的文件(那些已持久化到HFile中的Log可以删除)。WAL文件存储在/hbase/WALs/${HRegionServer_Name}的目录中
- BlockCache：它是一个读缓存，即“引用局部性”原理。
- HRegion：它是一个Table在一个RegionServer中的存储单元，也是分布式存储的最小单元。一个Table可以有一个或多个Region，他们可以在一个相同的RegionServer上，也可以分布在不同的RegionServer上，一个RegionServer可以有多个Region，他们分别属于不同的Table。Region由多个Store构成，每个Store对应了一个Table在这个Region中的一个ColumnFamily，即每个ColumnFamily就是一个集中的存储单元，因而最好将具有相近IO特性的Column存储在一个ColumnFamily，以实现高效读取(数据局部性原理，可以提高缓存的命中率)。

RegionServer组成

- HStore：HBase中存储的核心，它实现了读写HDFS功能，一个HStore由一个MemStore和0个或多个StoreFile组成。

- MemStore：是一个写缓存(InMemorySortedBuffer)，所有数据的写在完成WAL日志写后，再写入MemStore中，由MemStore根据一定的算法将数据Flush到地层HDFS文件中(HFile)，通常每个HRegion中的每个ColumnFamily有一个自己的MemStore。

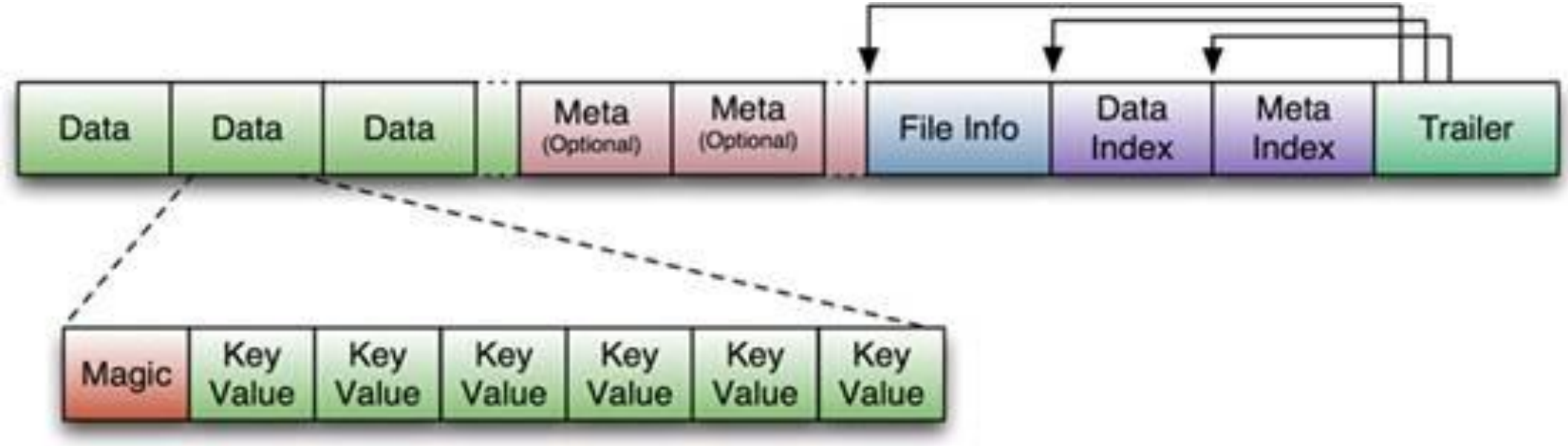
- HFile(StoreFile)：用于存储HBase的数据(Cell/KeyValue)。在HFile中的数据是按RowKey、ColumnFamily、Column排序，对相同的Cell(即这三个值都一样)，则按timestamp倒序排列。

Hfile存储格式

HBase中的所有数据文件都存储在Hadoop HDFS文件系统上，主要包括上述提出的两种文件类型：

- **HFile：** HBase中KeyValue数据的存储格式，HFile是Hadoop的二进制格式文件，实际上StoreFile就是对HFile做了轻量级包装，即StoreFile底层就是HFile
- **HLog File：**HBase中WAL（Write Ahead Log）的存储格式，物理上是Hadoop的Sequence File

下图是HFile的存储格式：



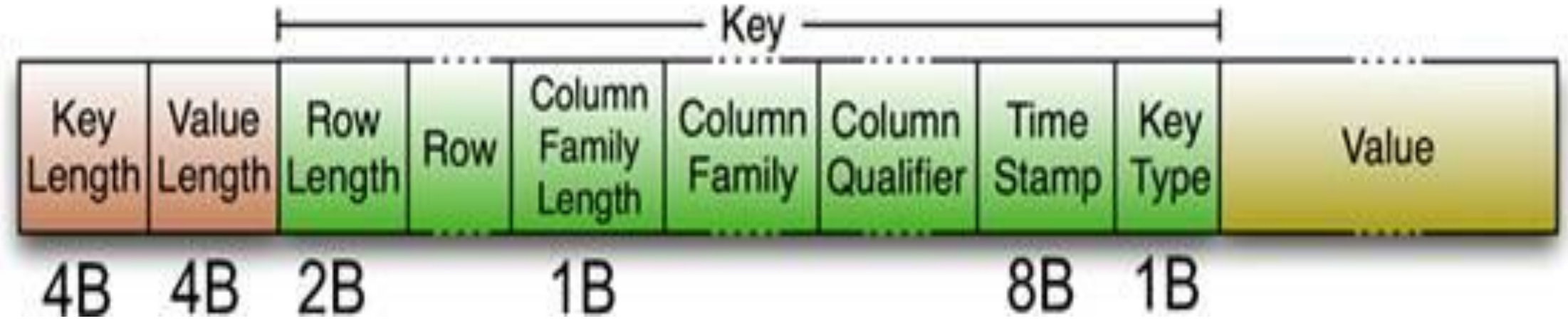
Hfile存储格式

- 首先HFile文件是不定长的，长度固定的只有其中的两块：Trailer和FileInfo。正如图中所示的，Trailer中有指针指向其他数据块的起始点。File Info中记录了文件的一些Meta信息，例如：AVG_KEY_LEN, AVG_VALUE_LEN, LAST_KEY, COMPARATOR, MAX_SEQ_ID_KEY等。Data Index和Meta Index块记录了每个Data块和Meta块的起始点。

- Data Block是HBase I/O的基本单元，为了提高效率，HRegionServer中有基于LRU的Block Cache机制。每个Data块的大小可以在创建一个Table的时候通过参数指定，大号的Block有利于顺序Scan，小号Block利于随机查询。每个Data块除了开头的Magic以外就是一个个KeyValue对拼接而成，Magic内容就是一些随机数字，目的是防止数据损坏。后面会详细介绍每个KeyValue对的内部构造。

Keyvalue格式

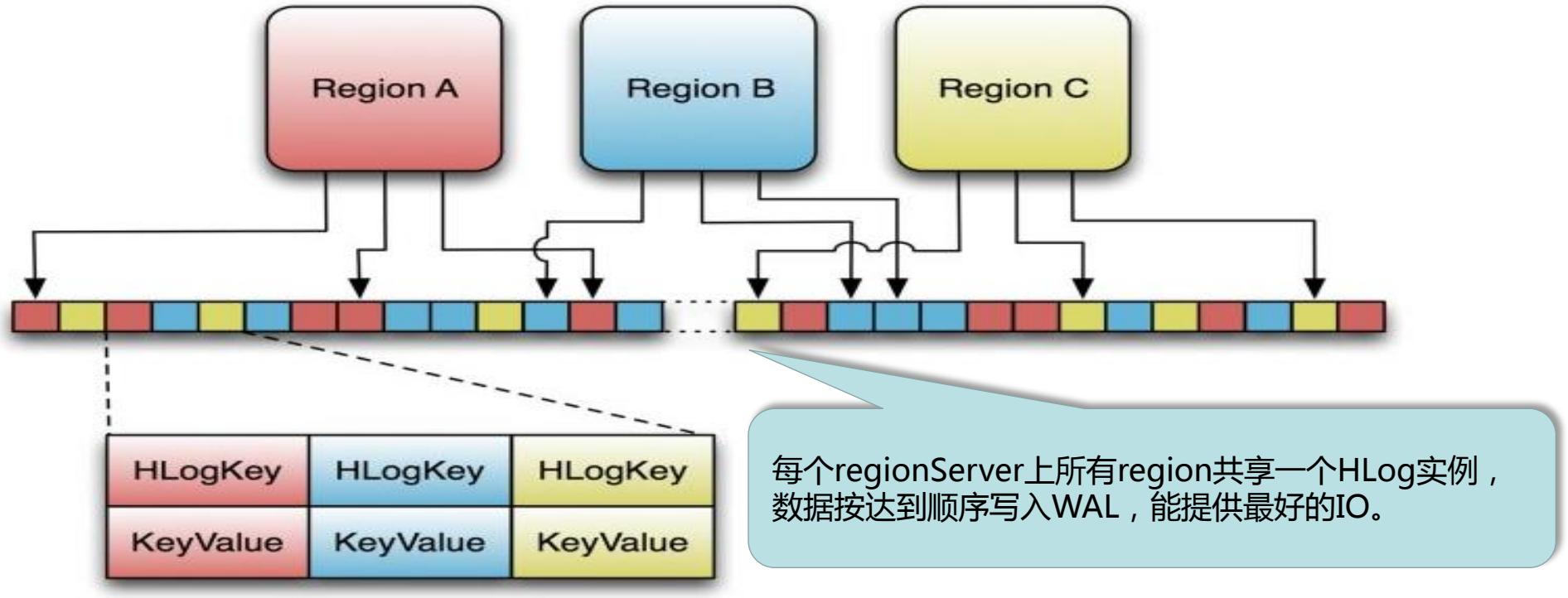
- HFile里面的每个KeyValue对就是一个简单的byte数组。但是这个byte数组里面包含了很多项，并且有固定的结构。我们来看看里面的具体结构：



- 开始是两个固定长度的数值，分别表示Key的长度和Value的长度。紧接着是Key，开始是固定长度的数值，表示RowKey的长度，紧接着是RowKey，然后是固定长度的数值，表示Family的长度，然后是Family，接着是Qualifier，然后是两个固定长度的数值，表示Time Stamp和Key Type（Put/Delete）。Value部分没有这么复杂的结构，就是纯粹的二进制数据了。

Hlog存储格式（HBase1.0前）

WAL(Write-Ahead-Log)：regionserver在处理插入和删除过程中用来记录操作内容的日志，只有日志写入成功，才会通知客户端操作成功。



Hlog存储格式（HBase1.0后）

MultiWAL：http://hbase.apache.org/book.html#hbase.versioning

70.6.2. MultiWAL

With a single WAL per RegionServer, the RegionServer must write to the WAL serially, because HDFS files must be sequential. This causes the WAL to be a performance bottleneck.

HBase 1.0 introduces support MultiWal in HBASE-5699. MultiWAL allows a RegionServer to write multiple WAL streams in parallel, by using multiple pipelines in the underlying HDFS instance, which increases total throughput during writes. This parallelization is done by partitioning incoming edits by their Region. Thus, the current implementation will not help with increasing the throughput to a single Region.

RegionServers using the original WAL implementation and those using the MultiWAL implementation can each handle recovery of either set of WALs, so a zero-downtime configuration update is possible through a rolling restart.

HLog存储格式

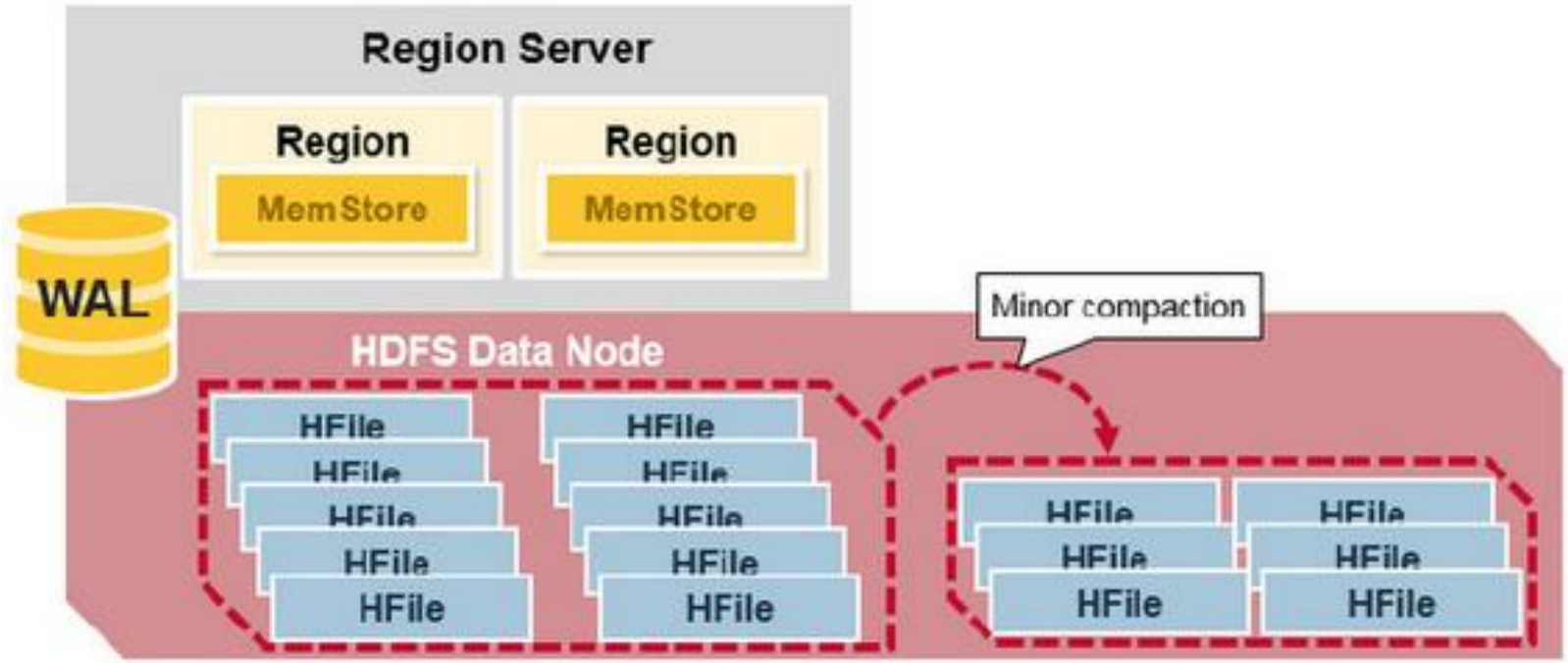
- 上图中示意了HLog文件的结构，其实HLog文件就是一个普通的Hadoop Sequence File，Sequence File 的Key是HLogKey对象，HLogKey中记录了写入数据的归属信息，除了table和region名字外，同时还包括 sequence number和timestamp，timestamp是“写入时间”，sequence number的起始值为0，或者是最近一次存入文件系统中sequence number。

- HLog Sequece File的Value是HBase的KeyValue对象，即对应HFile中的KeyValue，可参见上文描述。

Region-StoreFile小合并

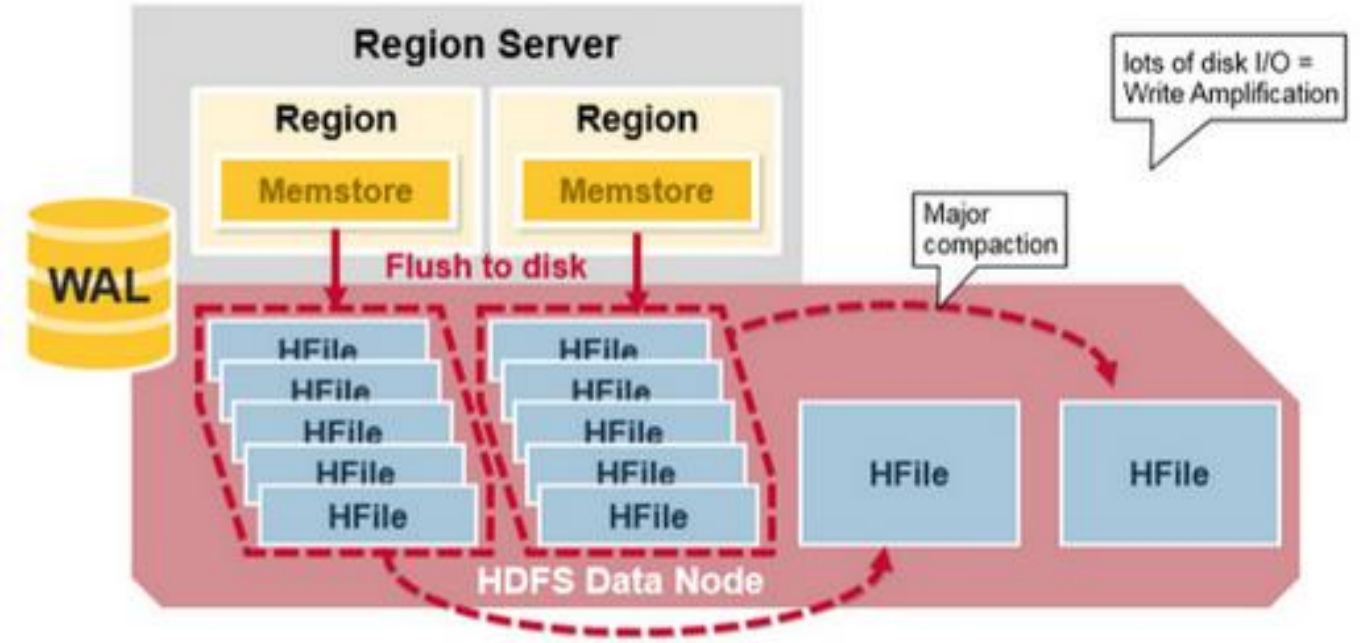
MemStore每次Flush会创建新的HFile，而过多的HFile会引起读的性能问题，Hbase采用Compaction机制来解决这个问题，Compaction分为两种：Minor Compaction和Major Compaction。

Minor Compaction: 是指选取一些小的、相邻的StoreFile将他们合并成一个更大的StoreFile，在这个过程中不会处理已经Deleted或Expired的Cell。一次Minor Compaction的结果是更少并且更大的StoreFile



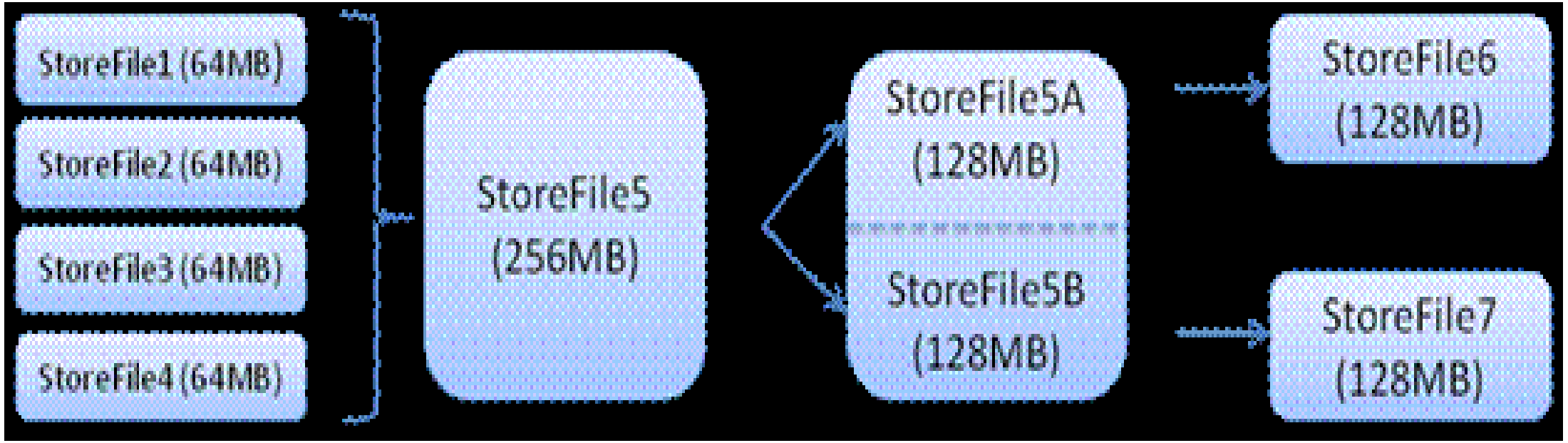
Region-StoreFile大合并

Major Compaction: 是指将所有的StoreFile合并成一个StoreFile，在这个过程中，标记为Deleted的Cell会被删除，而那些已经Expired的Cell会被丢弃，那些已经超过最多版本数的Cell会被丢弃。一次Major Compaction的结果是一个HStore只有一个StoreFile存在，Major Compaction可以手动或自动触发，然而由于它会引起很多的IO操作而引起性能问题，因而它一般会被安排在周末、凌晨等集群比较闲的时间。

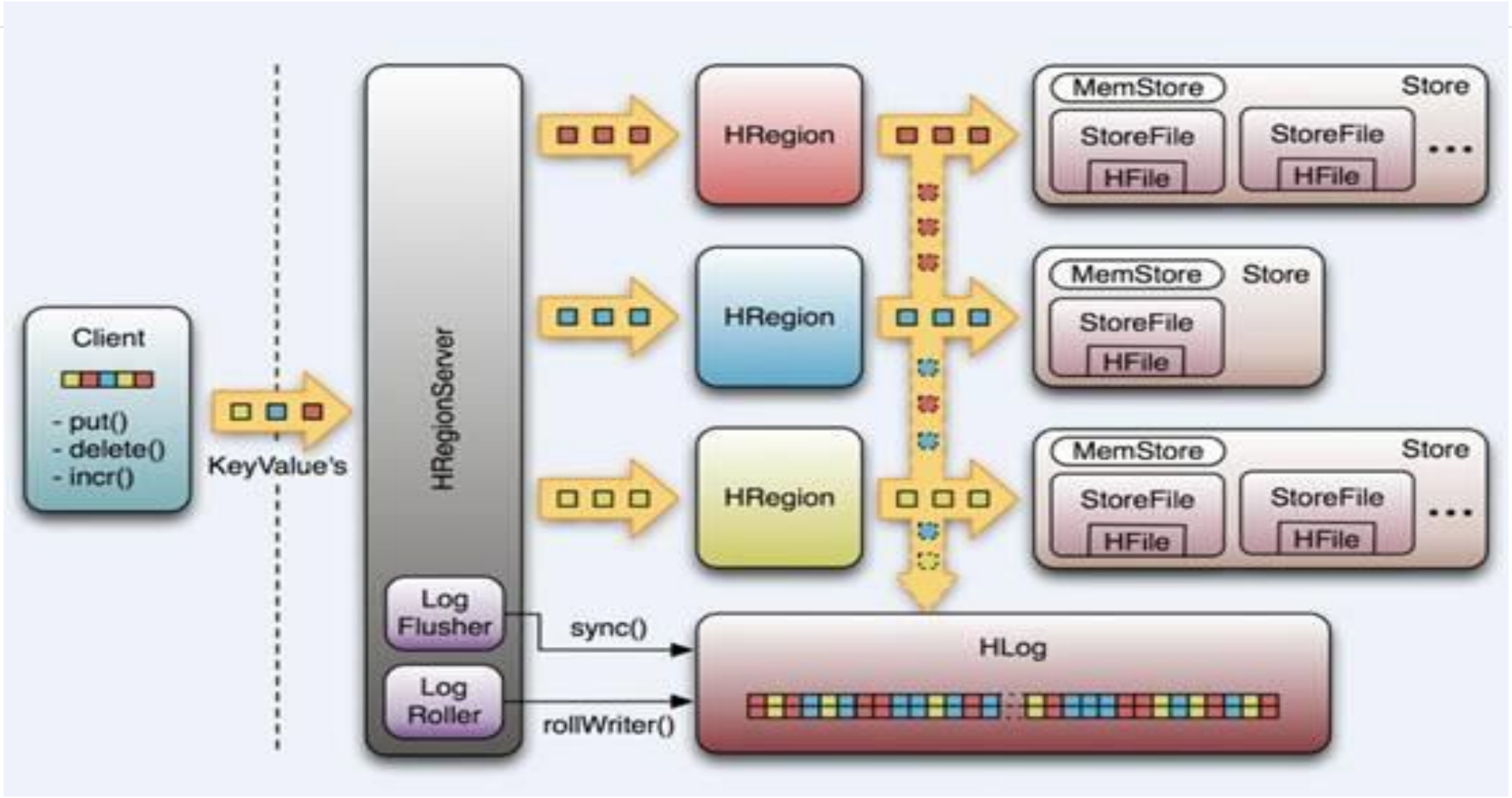


HBase Split拆分

最初，一个Table只有一个HRegion，随着数据写入增加，如果一个HRegion到达一定的大小，就需要Split成两个HRegion，这个大小由hbase.hregion.max.filesize指定split时，两个新的HRegion会在同一个HRegionServer中创建，它们各自包含父HRegion一半的数据，当Split完成后，父HRegion会下线，而新的两个子HRegion会向HMaster注册上线；处于负载均衡的考虑，这两个新的HRegion可能会被HMaster分配到其他的HRegionServer。



Hbase Write-Ahead-Log（预先写日志）

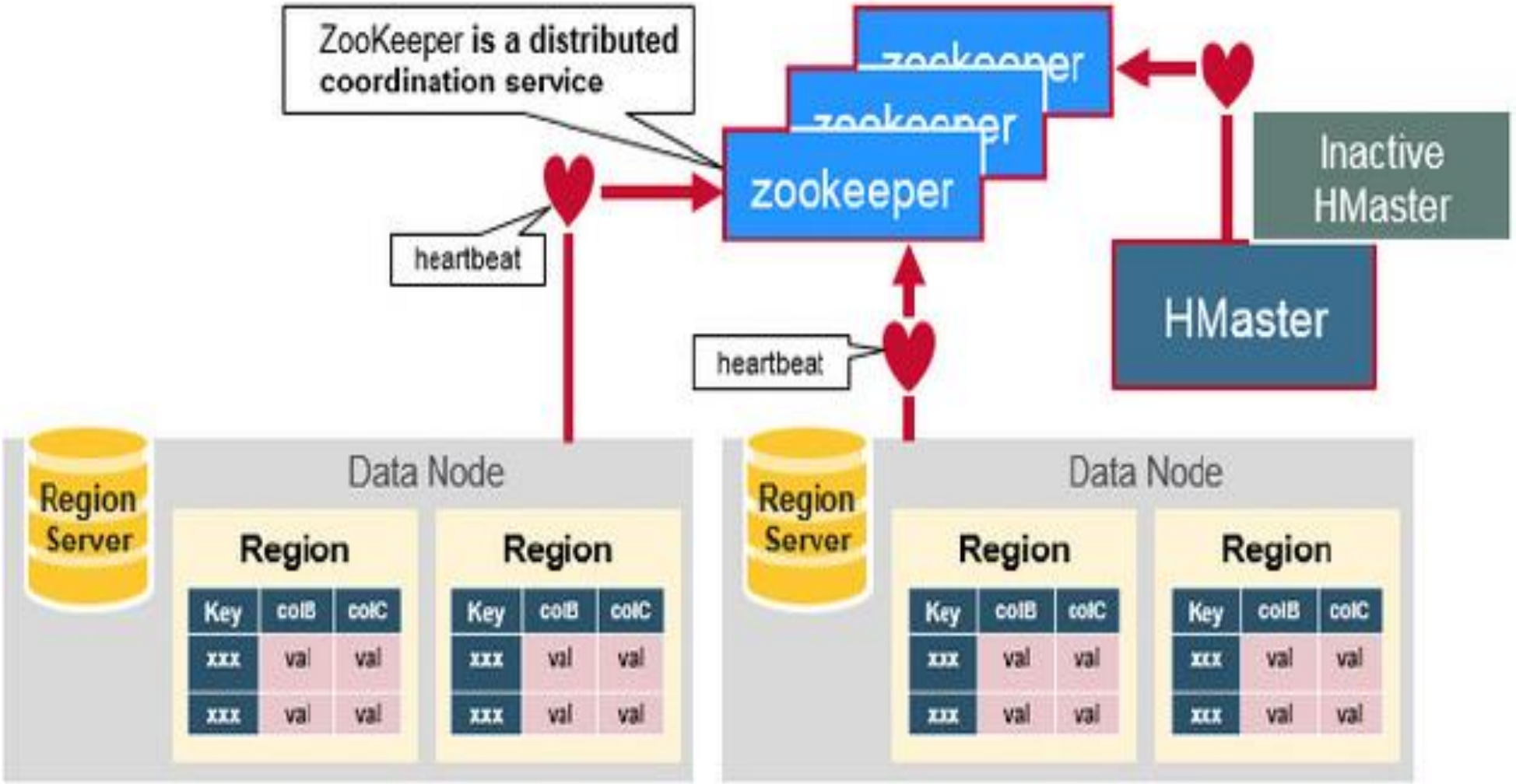


HLog Replay

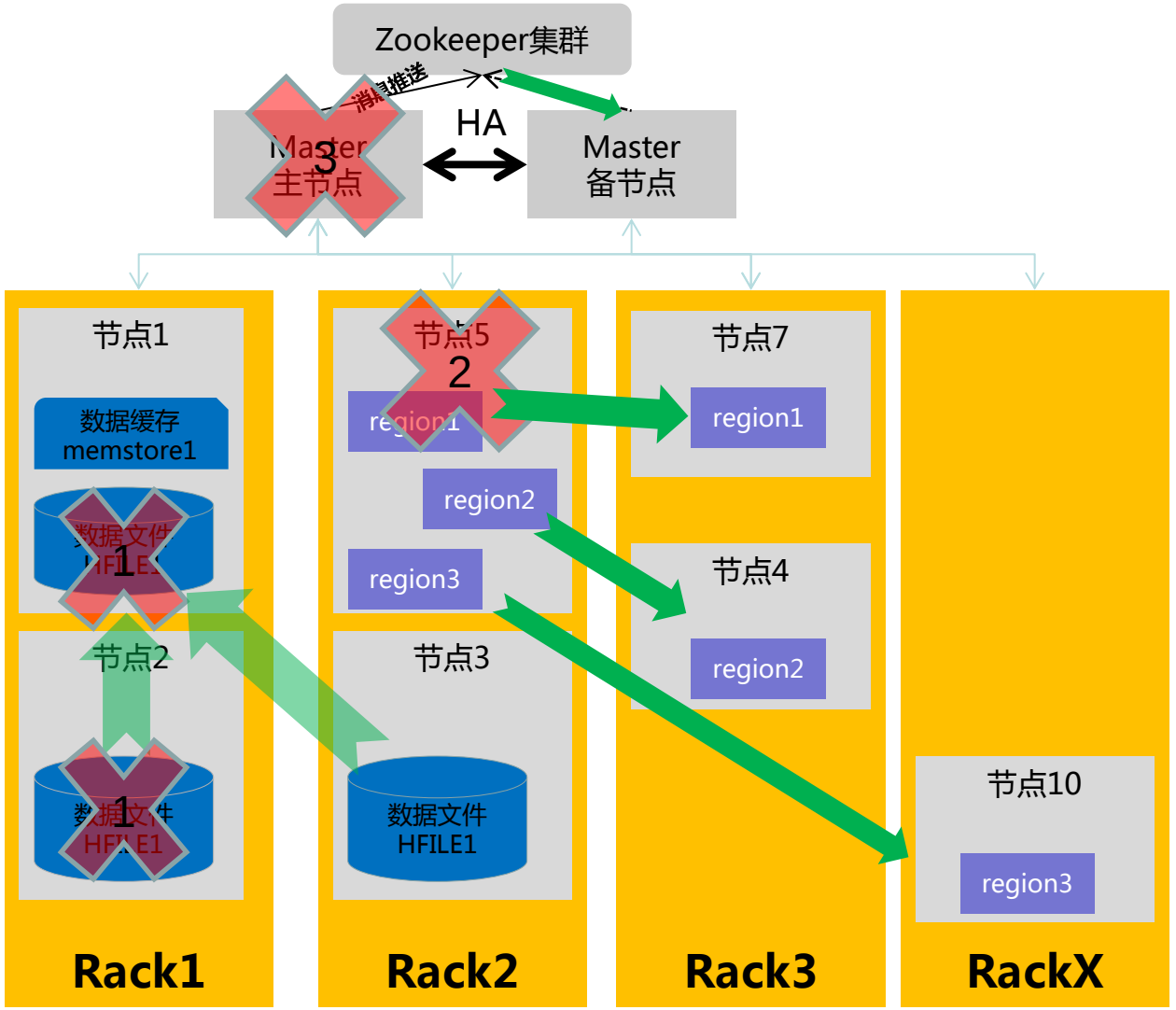
在分布式系统环境中，无法避免系统出错或者宕机，因此一旦HRegionServer意外退出，MemStore中的内存数据将会丢失，这就需要引入HLog了。每个HRegionServer中都有一个HLog对象，HLog是一个实现Write Ahead Log的类，在每次用户操作写入MemStore的同时，也会写一份数据到HLog文件中（HLog文件格式见后续），HLog文件定期会滚动出新的，并删除旧的文件（已持久化到StoreFile中的数据）。

当HRegionServer意外终止后，HMaster会通过Zookeeper感知到，HMaster首先会处理遗留的 HLog文件，将其中不同Region的Log数据进行拆分，分别放到相应region的目录下，然后再将失效的region重新分配，领取 到这些region的HRegionServer在Load Region的过程中，会发现有历史HLog需要处理，因此会Replay HLog中的数据到MemStore中，然后flush到StoreFiles，完成数据恢复。

HBase高可用



HBase高可用



1 **HDFS机架识别策略**

✓当数据文件损坏时，会找相同机架上备份的数据文件，如果相同机架上的数据文件也损坏会找不同机架备份数据文件。

2 **HBASE的REGION快速恢复**

✓当节点损坏时，节点上的丢失的region，会在其他节点上均匀快速恢复。

3 **Master节点的HA机制**

✓Master为一主多备，当Master主节点宕机后，剩下的备节点通过选举，产生主节点。

HBase容错性

- **Master容错：Zookeeper重新选择一个新的Master**

  无Master过程中，数据读取仍照常进行；

  无master过程中，region切分、负载均衡等无法进行；

- **RegionServer容错：定时向Zookeeper汇报心跳，如果一旦时间内未出现心跳**

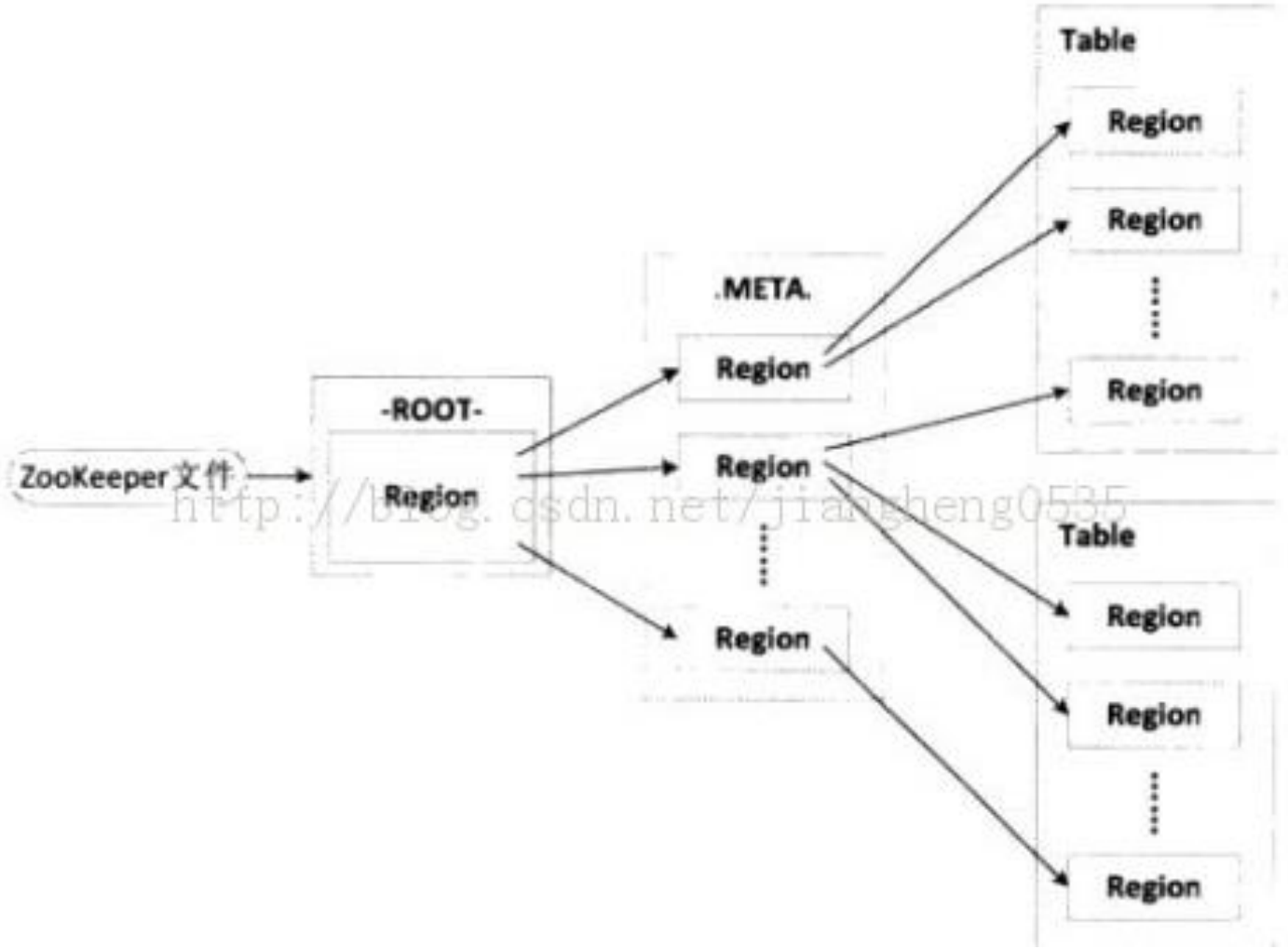
  Master将该RegionServer上的Region重新分配到其他RegionServer上；

  失效服务器上“预写”日志由主服务器进行分割并派送给新的RegionServer

- **Zookeeper容错：Zookeeper是一个可靠地服务**

  一般配置3或5个Zookeeper实例

Region定位



- 寻找RegionServer
  - ✓ ZooKeeper
  - ✓ -ROOT-(单Region)
  - ✓ .META.
  - ✓ 用户表

-ROOT-和.META.表结构

RowKey	info			historian
	regioninfo	server	serverstartcode	
TableName,StartKey,TimeStamp	StartKey, EndKey, Family List { Family,BloomFilter,Compress,TTL,InMemory,BlockSize,BlockCache}	address		

-ROOT-表和.META.

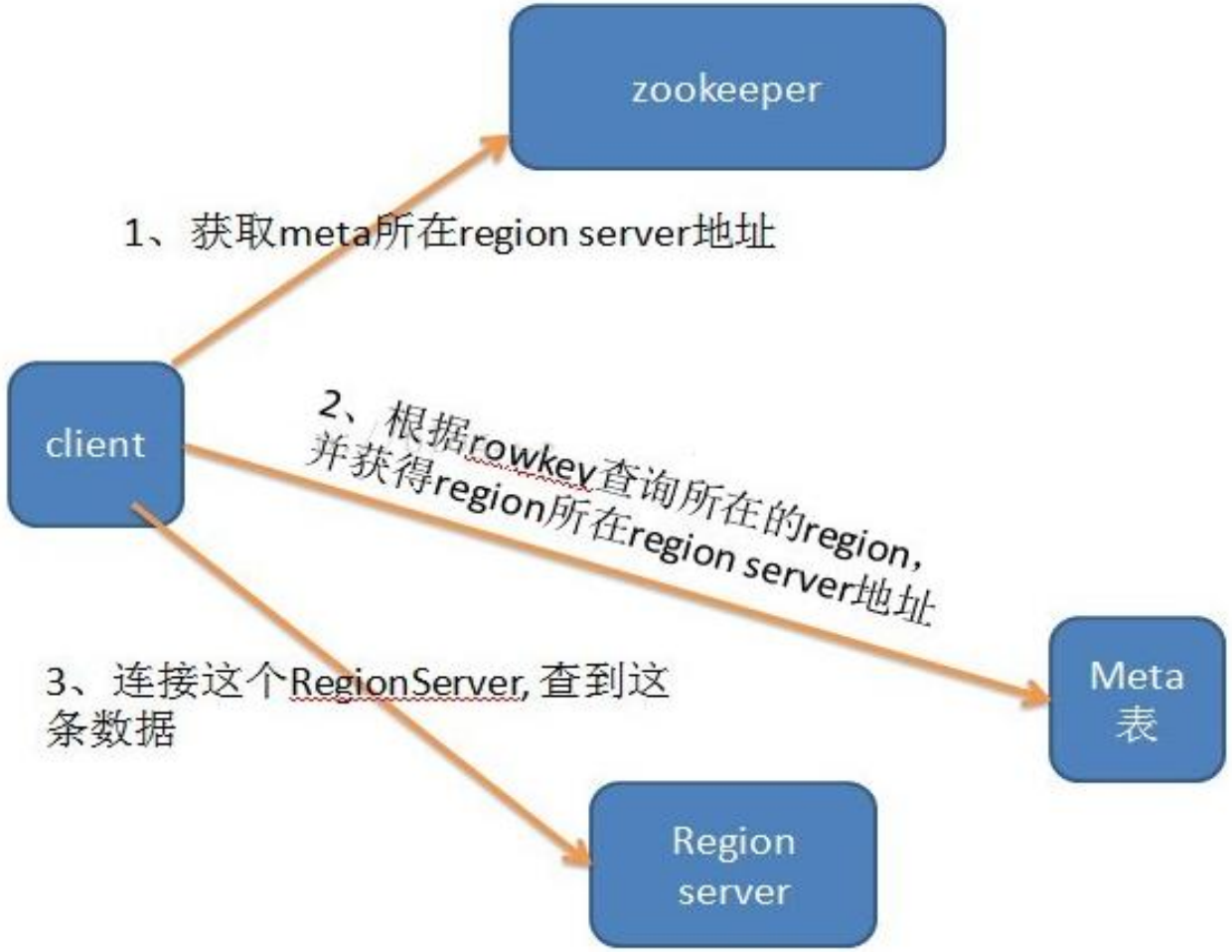
- -ROOT-
  - ✓表包含.META.表所在的region列表，该表只会有一个Region；
  - ✓Zookeeper中记录了-ROOT-表的location。

- .META.
  - ✓ 表包含所有的用户空间region列表，以及RegionServer的服务器地址。

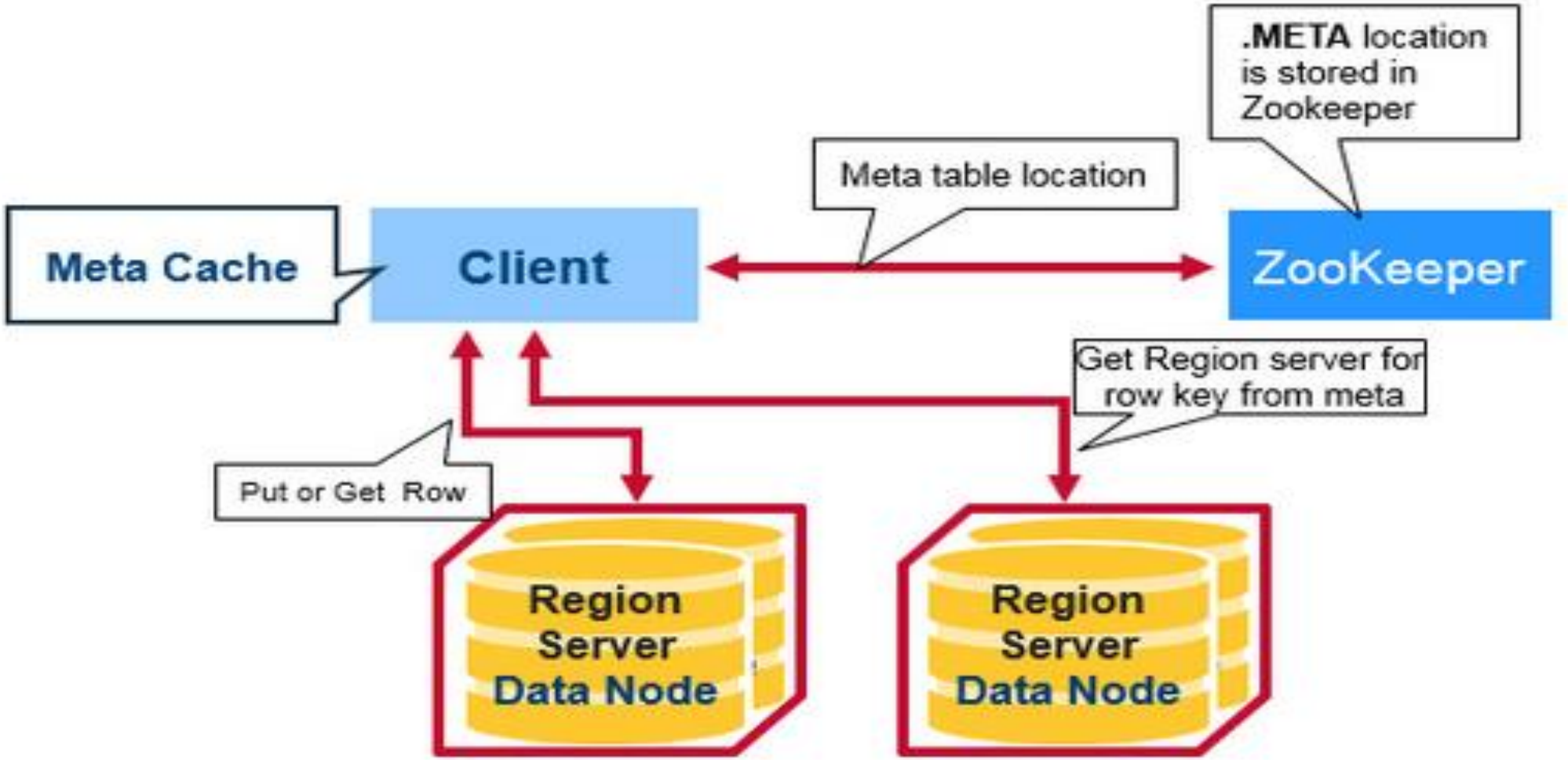
Region定位改进.

- Region定位优化改进
  - ✓ 在hbase 0.96 之前的版本中，hbase catalog包括-ROOT-和.META.两个表，但是在0.96以及之后的hbase版本中，-ROOT-表被废弃，只有hbase:meta（即.META.）。HBase的meta信息存储在表hbase:meta中，该表也是一个hbase表，但是在hbase shell下执行list命令时，会将该表过滤掉，不会显示。
  - ✓ 在hbase:meta中，存储着所有regions的信息，且该表的位置直接记录在zookeeper中，而无需-ROOT-表。

Region定位改进

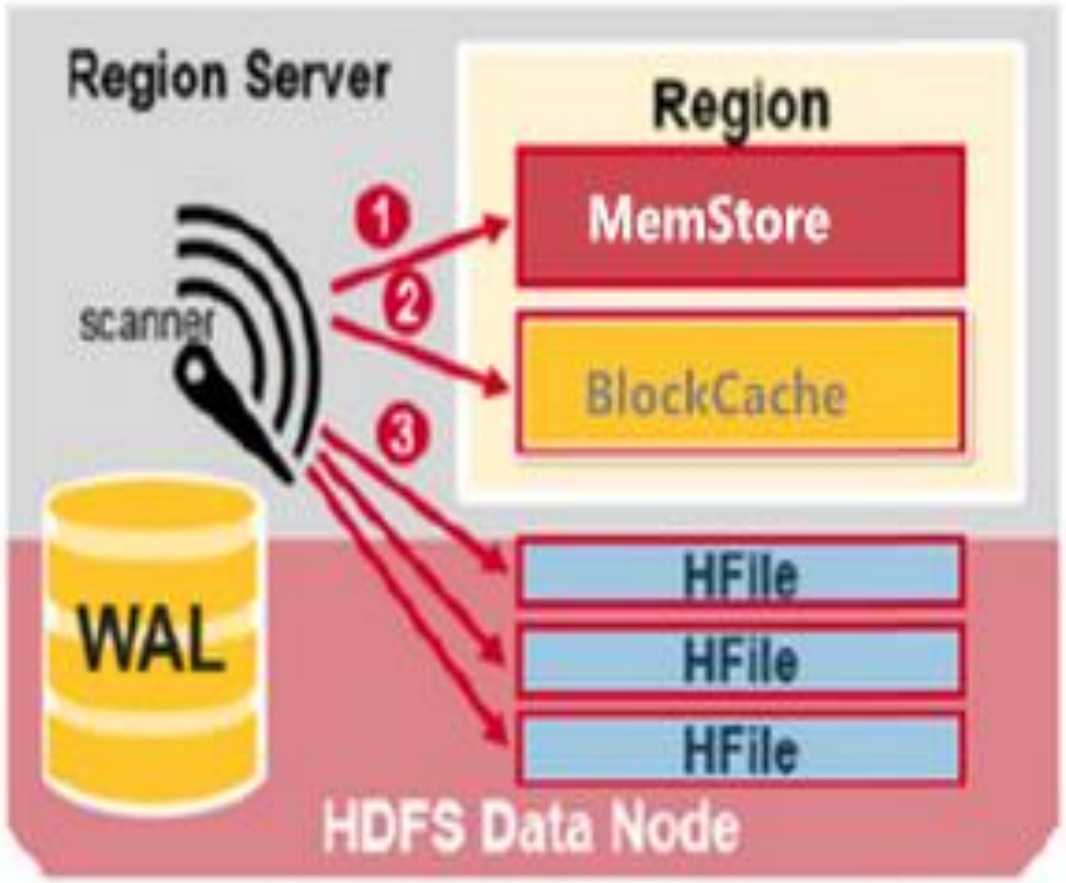


Hbase 读流程



Hbase 读流程

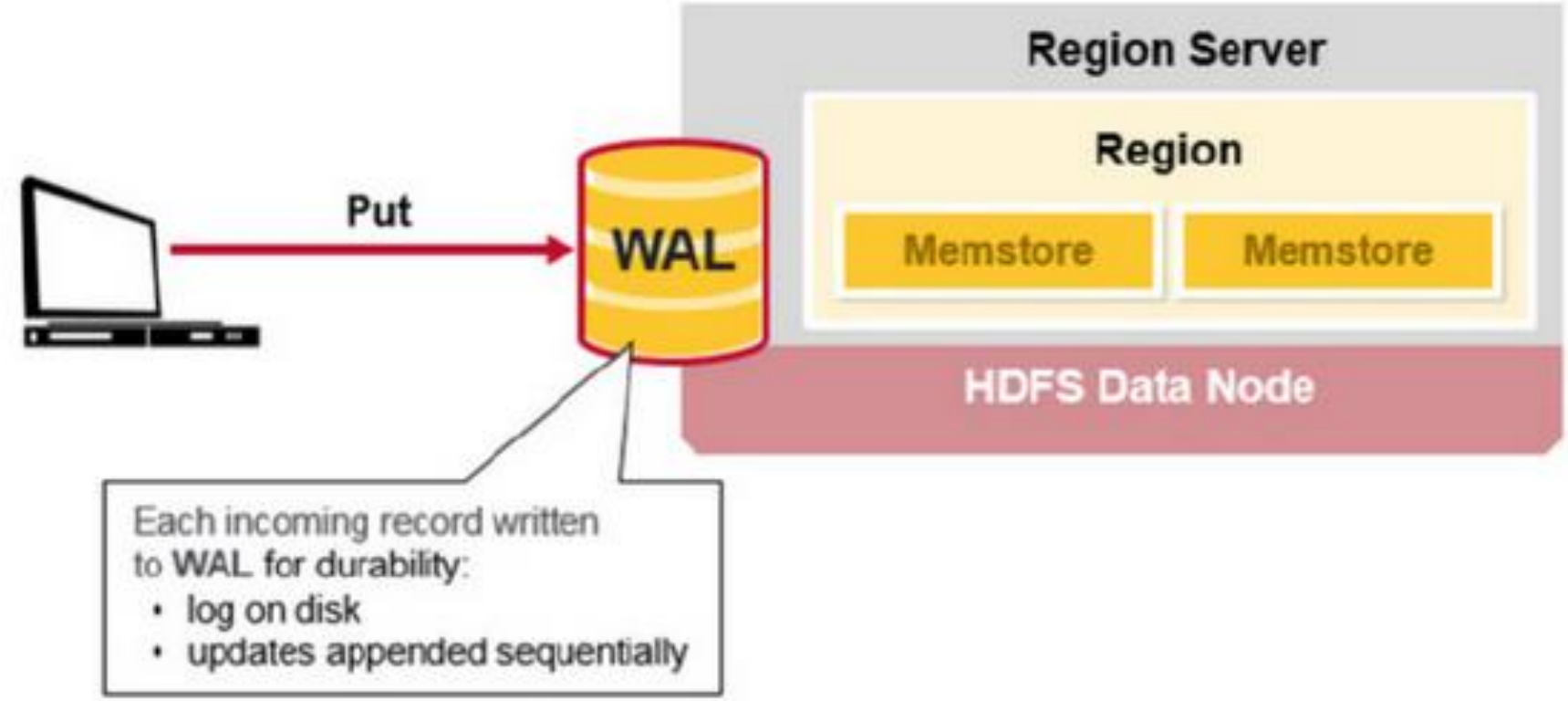
1. First the scanner looks for the Row KeyValues in the Memstore
2. Next the scanner looks in the BlockCache
3. If all row cells not in MemStore or blockCache, look in HFiles



HBase put写流程

- 当客户端发起一个Put请求时，首先根据RowKey寻址，从hbase:meta表中查出该Put数据最终需要去的HRegionServer

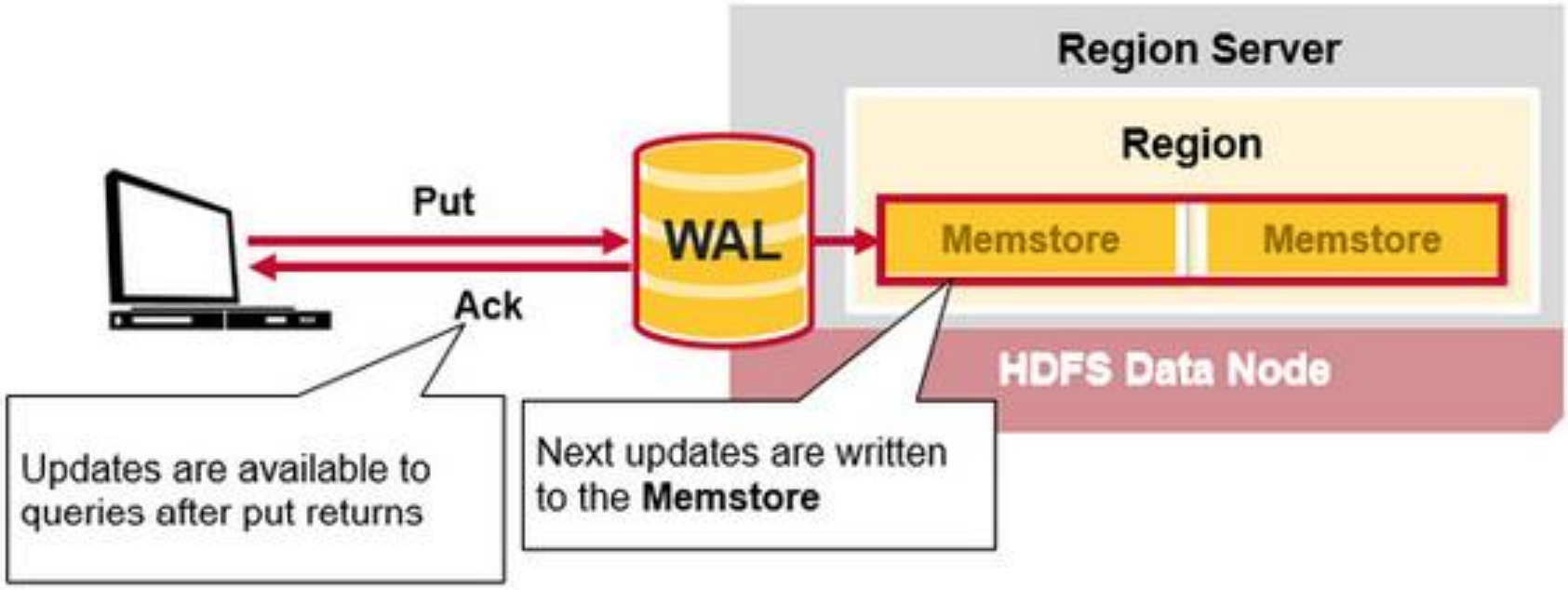
- 客户端将Put请求发送给相应的HRegionServer，在HRegionServer中它首先会将该Put操作写入WAL日志文件中(Flush到磁盘中)



HBase put写流程

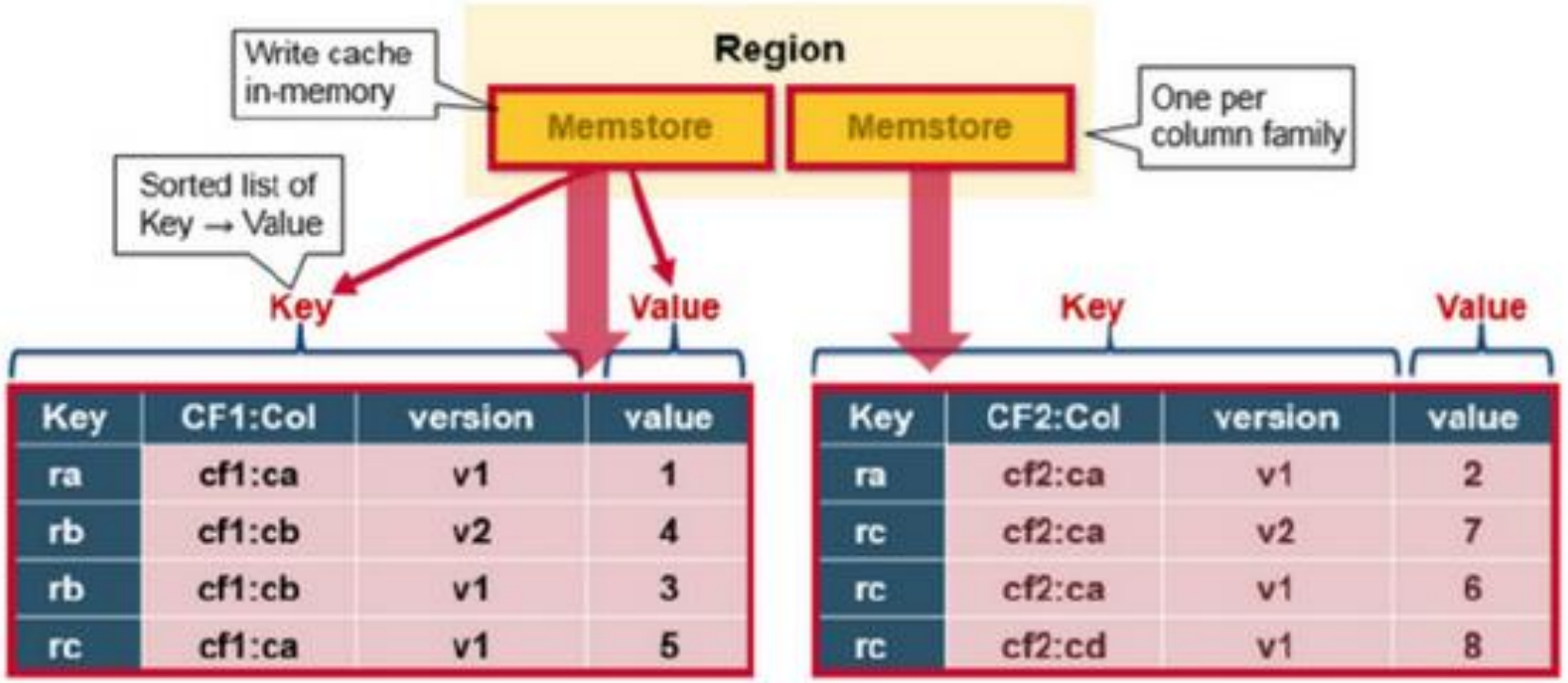
- 写完WAL日志文件后，HRegionServer根据Put中的TableName和RowKey找到对应的HRegion，并根据Column Family找到对应的HStore

- 将Put数据写入到该HStore的MemStore中。此时写成功，并返回通知客户端



HBase put写流程

➢ MemStore是一个In Memory Sorted Buffer，在每个HStore中都有一个MemStore，即它是一个HRegion的一个Column Family对应一个实例，它的排列顺序以RowKey、Column Family、Column的顺序以及Timestamp的倒序

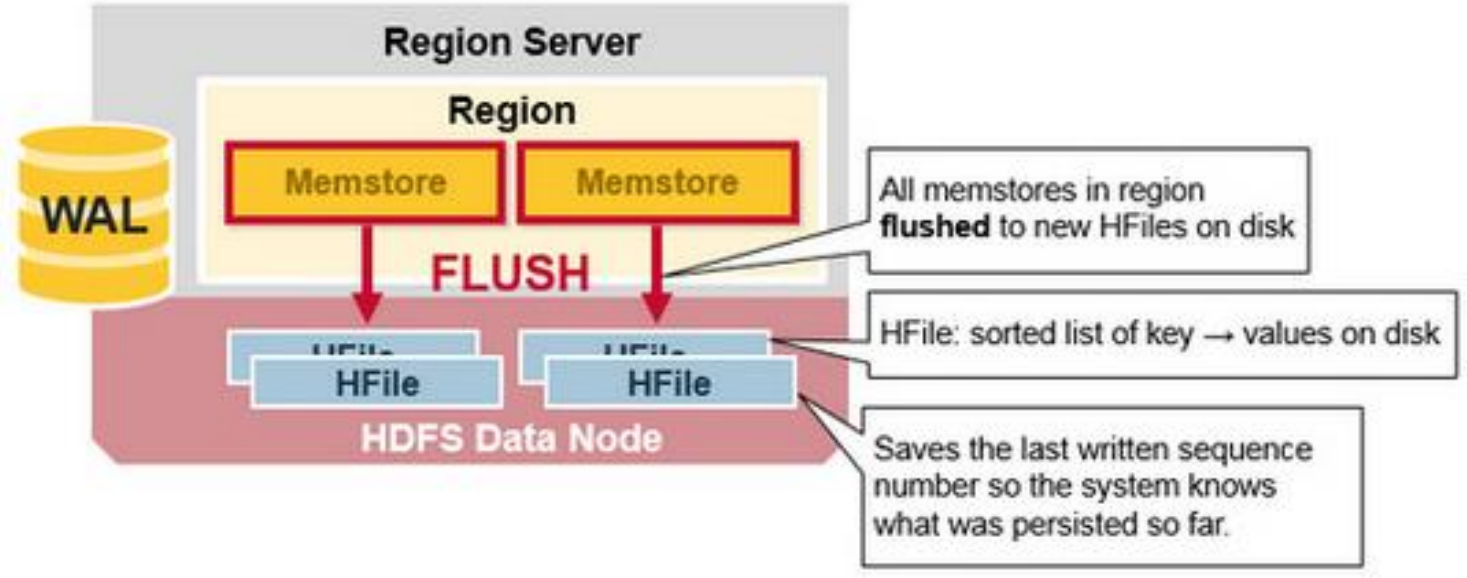


Key	CF1:Col	version	value
ra	cf1:ca	v1	1
rb	cf1:cb	v2	4
rb	cf1:cb	v1	3
rc	cf1:ca	v1	5

Key	CF2:Col	version	value
ra	cf2:ca	v1	2
rc	cf2:ca	v2	7
rc	cf2:ca	v1	6
rc	cf2:cd	v1	8

HBase put写流程

- 每一次Put请求都是先写入到MemStore中，当MemStore满后会Flush成一个新的StoreFile(底层实现是HFile)，即一个HStore(ColumnFamily)可以有0个或多个StoreFile(HFile)
  注意：MemStore的最小Flush单元是HRegion而不是单个MemStore， 这就是建议使用单列族的原因，太多的Column Family一起Flush会起性能问题。
- 在MemStore Flush过程中，还会在尾部追加一些meta数据，其中就包括Flush时最大的WAL sequence值，以告诉HBase这个StoreFile写入的最新数据的序列，那么在Recover时就知道从哪里开始。在HRegion启动时，这个sequence会被读取，并取最大的作为下一次更新时的起始sequence。



07 HBase与其它系统比较

HBase vs HDFS

- 两者都具有良好的容错性和扩展性，都可以扩展到成百上千个节点

- HDFS适合批处理场景

  ✓不支持随机随机查找（全文件扫描）
  ✓不适合增量数据处理（处理整个文件，不能处理其中一部分）
  ✓不支持数据更新（一旦存储，无法修改）

- HBase是对HDFS很好的补充
  ✓ 可以插入一条数据
  ✓可以删除一条数据
  ✓可以随机查询、过滤数据

Hbase VS 关系型数据库

	关系型数据库	HBase
数据存储	面向行	面向列
事务	支持多行事务	仅支持单行事务
查询语言	Sql	get、put、scan
安全	比较完善	欠缺
索引	对任意列建立索引	只支持row key
数据大小	TB	PB级别以上数据
读写吞吐率	适合查询少量数据	适合批量数据查询

Hbase VS Hive

	Hive	HBase
联系	hbase与hive都是架构在hadoop之上的。都是用hadoop作为底层存储	
区别	1、Hive中的表是纯逻辑表，只是表的定义等，即表的元数据。Hive本身不存储数据，它完全依赖HDFS和MapReduce。这样就可以将结构化的数据文件映射为为一张数据库表，并提供完整的SQL查询功能，并将SQL语句最终转换为MapReduce任务。 2、Hive是基于MapReduce来处理数据,而MapReduce处理数据是基于行的模式 3、Hive表是稠密型，即定义多少列，每一行有存储固定列数的数据。 4、Hive使用Hadoop来分析处理数据，而Hadoop系统是批处理系统，因此不能保证处理的低迟延问题 5、Hive不提供row-level的更新，它适用于大量append-only数据集（如日志）的批任务处理。 6、Hive提供完整的SQL实现，通常被用来做一些基于历史数据的挖掘、分析。	1、HBase表是物理表，适合存放非结构化的数据。 2、HBase处理数据是基于列的而不是基于行的模式，适合海量数据的随机访问。 3、HBase的表是疏松的存储的，因此用户可以给行定义各种不同的列 4、HBase是近实时系统，支持实时查询。 5、HBase的查询，支持和row-level的更新。 6、HBase不适用与有join，多级索引，表关系复杂的应用场景。

Hbase VS Hive

- 存储同样是hdfs，同样有自己的数据表，为什么Hbase比Hive查询快那么多？

➢ Hive

1、hive 是基于MapReduce来处理，离线计算速度慢。
2、MapReduce处理数据是基于行的模式，查询数据默认是扫描整个表。
3、hive中的表是纯逻辑表，只是表的定义，本身不存储数据，完全依赖hdfs和MapReduce。

➢ Hbase

1、hbase 是物理表，有独立的物理结构，查询的时候将相应的数据加载到内存，提升后续的查询效率。
2、Hbase 是基于列存储的，查询的时候可以只扫描某一个列或者某几列，避免扫描所有数据。
3、Hbase 有一级索引rowkey，根据rowkey查询速度非常快。
4、Hbase 提供了一个超大的内存hash表，搜索引擎通过这个hash表存储索引，提升查询效率。

大讲台
dajiangtai.com

THANKS