HBase

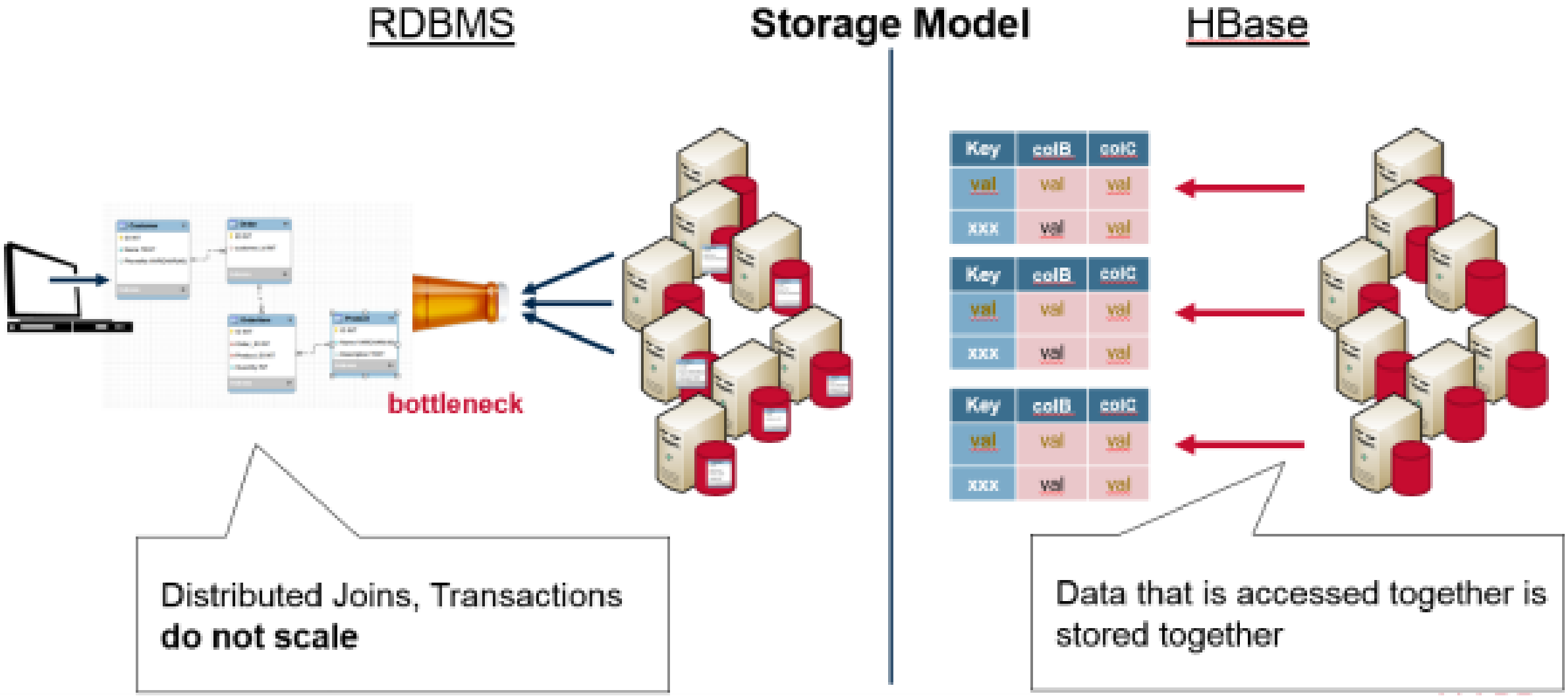
nlslzf@126.com
QQ:122609030
wechat:aiya_pxw
On Hbase 1.2
V0.1 2016.Q4 囫囵吞枣 独上高楼望尽天涯路

目录

- 核心架构
- 参数调优
- 运维命令

核心架构：概念模型

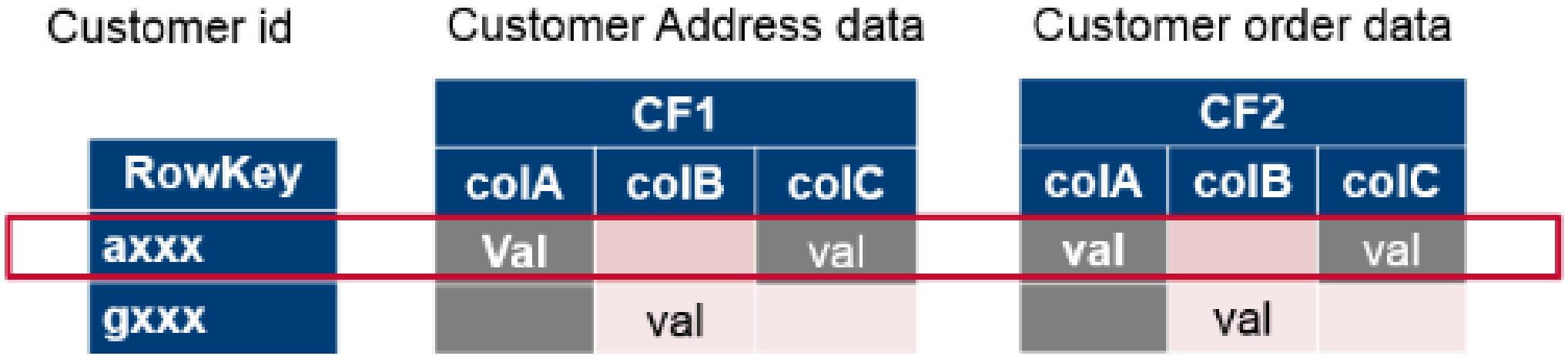
Relational databases vs. HBase - data storage model



SQL数据库无法水平扩展，分布式情况下，传统数据库的join操作和事务管理都无法实现

HBase Designed for Distribution, Scale, and database

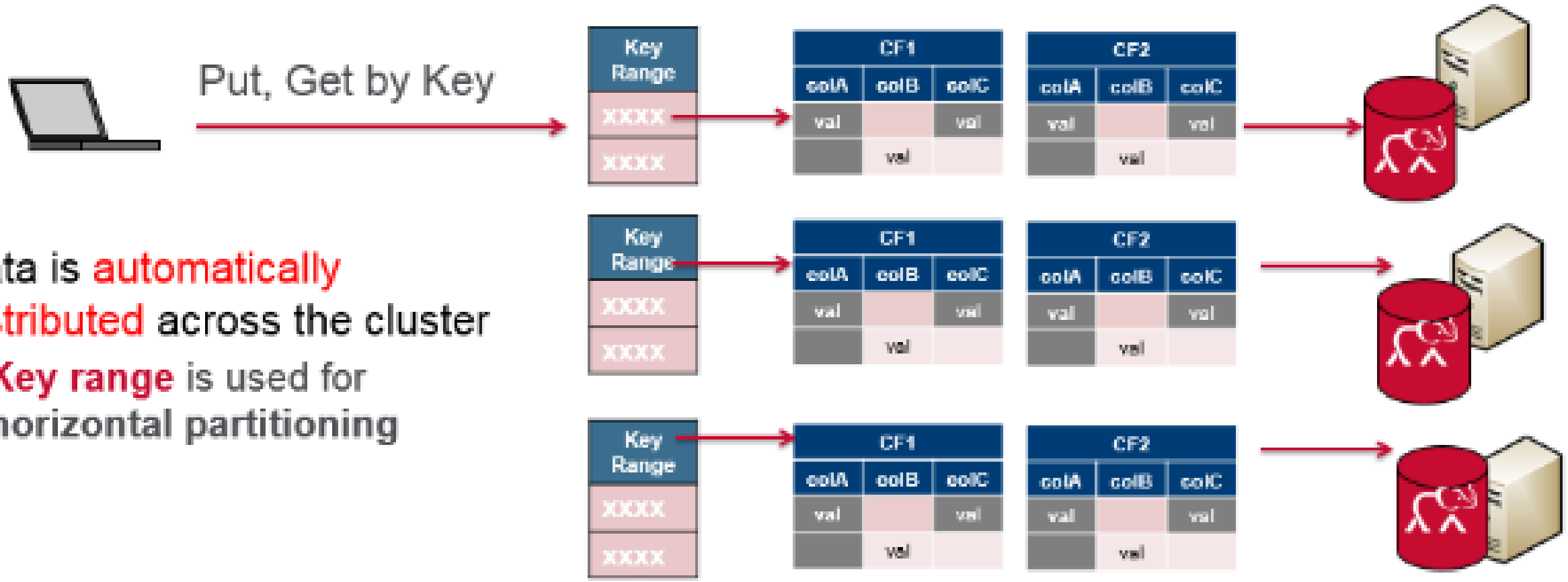
SpeedHBase is a column family-oriented



Data is accessed and stored together:

- RowKey is the primary index
- Column Families group similar data by row key

HBase is a distributed database



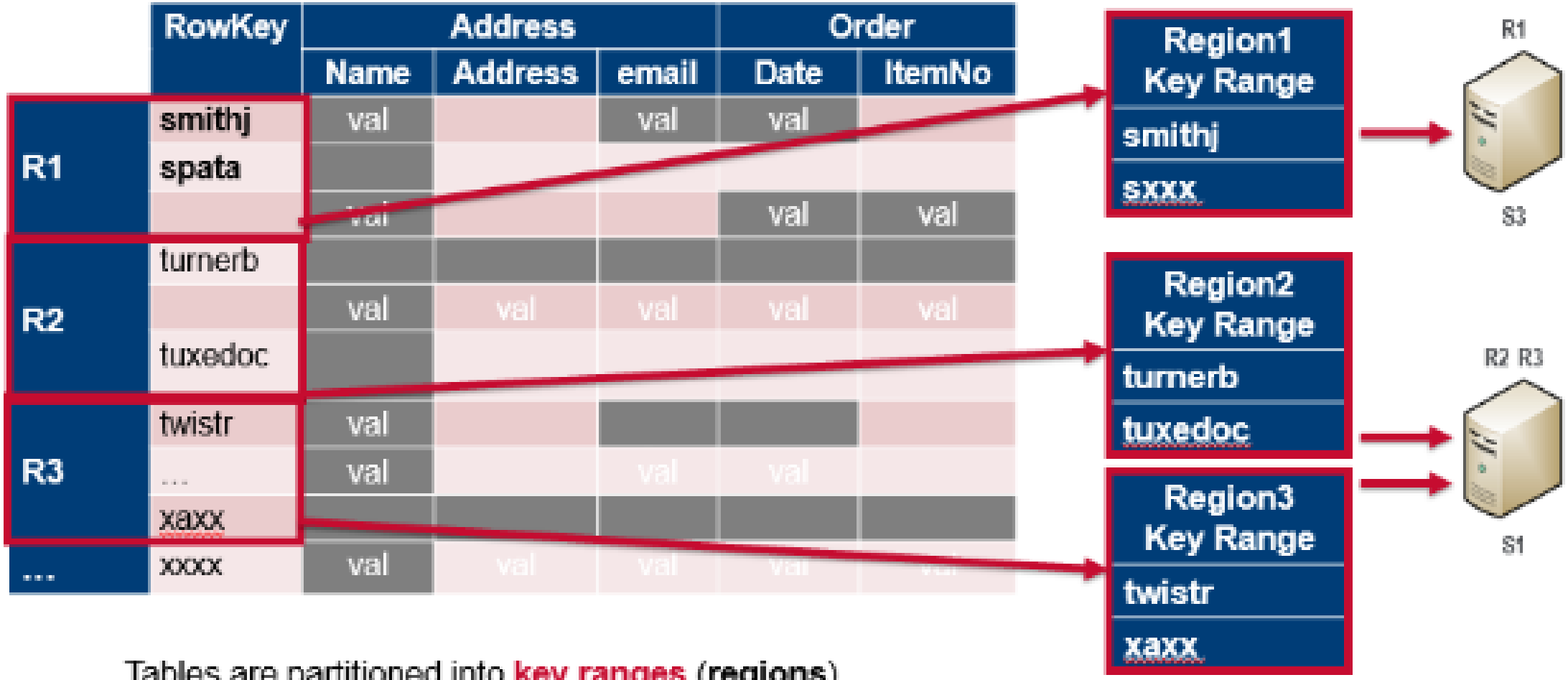
Data is automatically distributed across the cluster

- **Key range** is used for horizontal partitioning

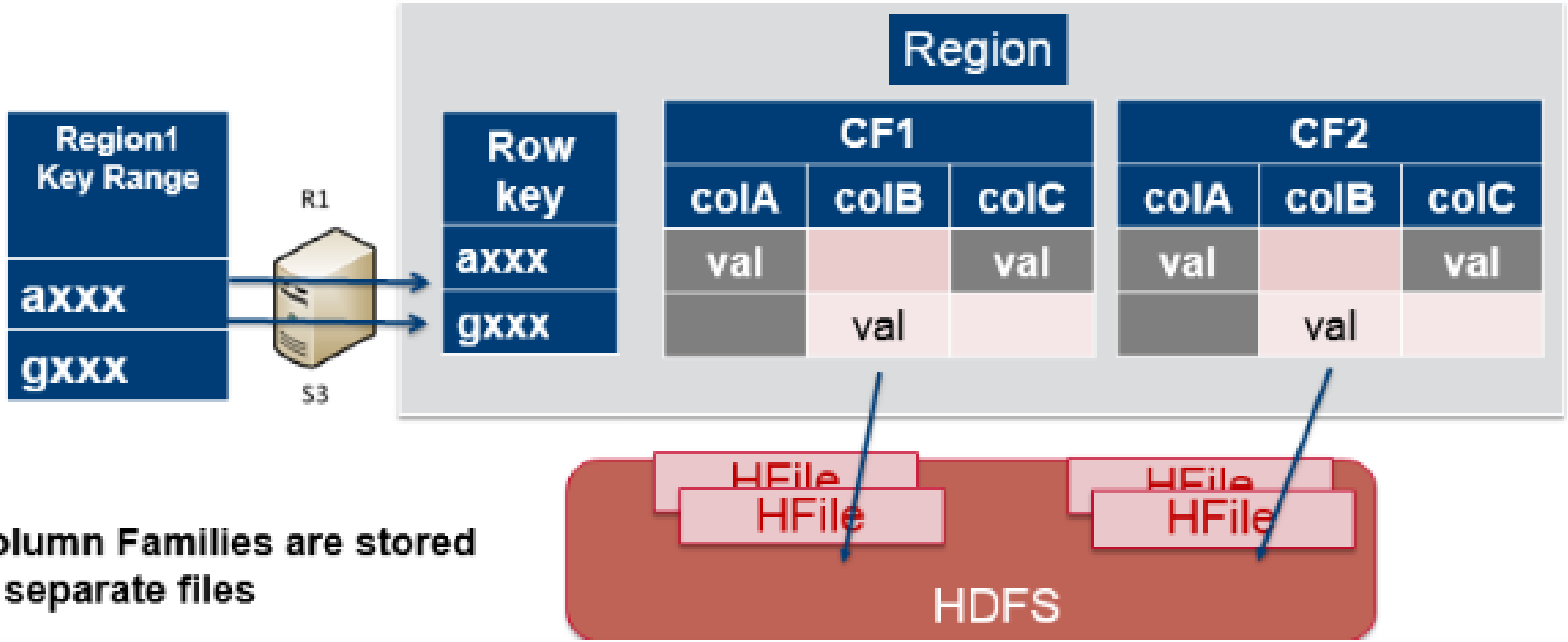
HBase data model – row keys

RowKey	Address			Order				...
	street	city	state	Date	ItemNo	Ship Address	Cost	
smithj	val		val	val			val	
spata								
sxxxx	val			val	val	val		
...								
turnerb	val	val	val	val	val	val	val	
...								
	val							
twistr	val		val	val			val	
...								
zaxx	val	val	val	val	val	val		
zxxx	val						val	

Tables are split into regions = contiguous keys

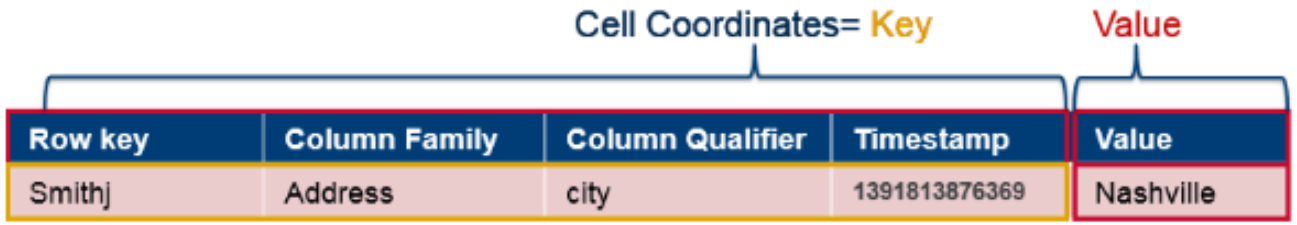


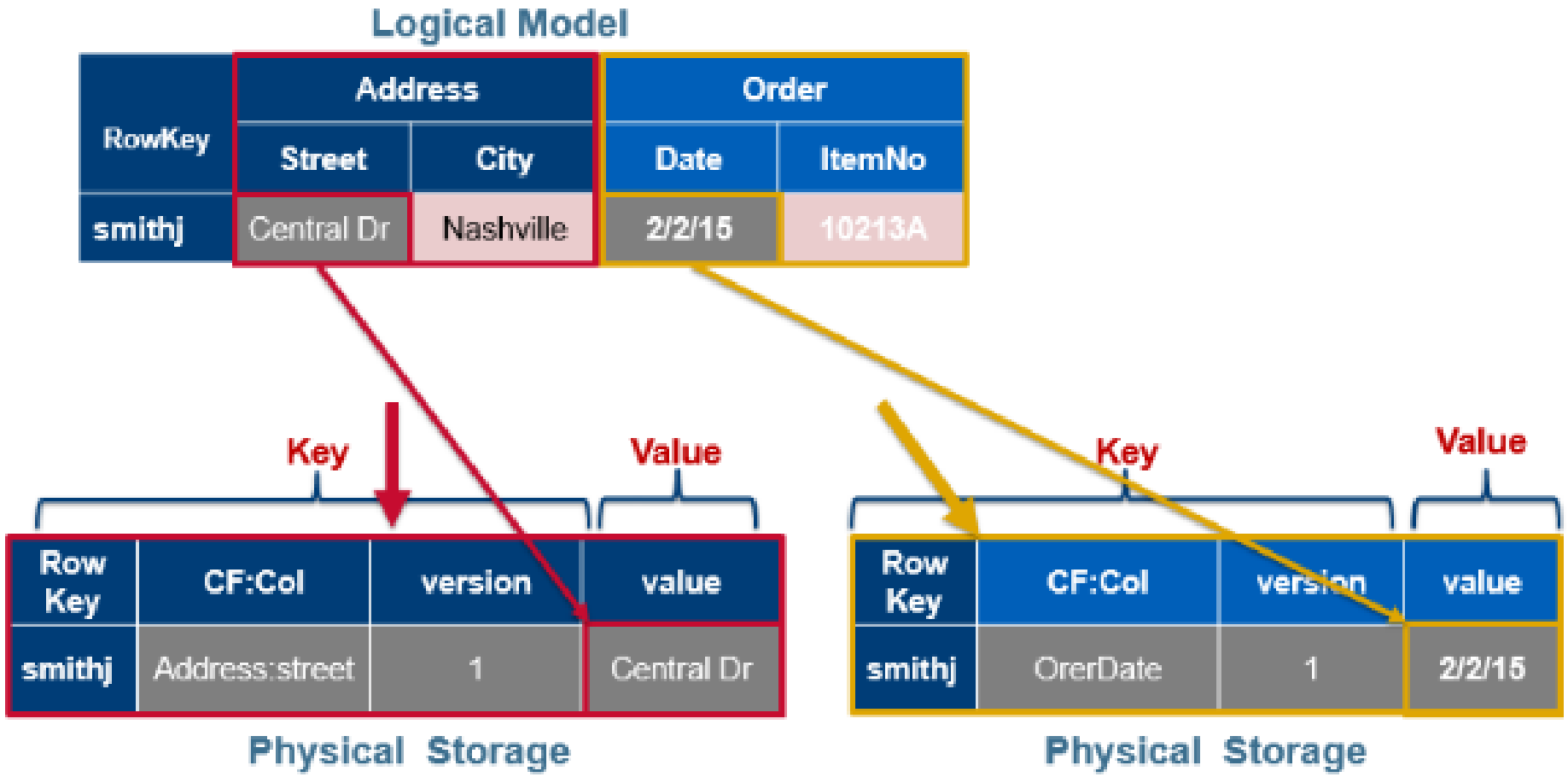
Tables are partitioned into **key ranges** (**regions**)
Region= served by nodes (Region Servers)
Regions are spread across cluster



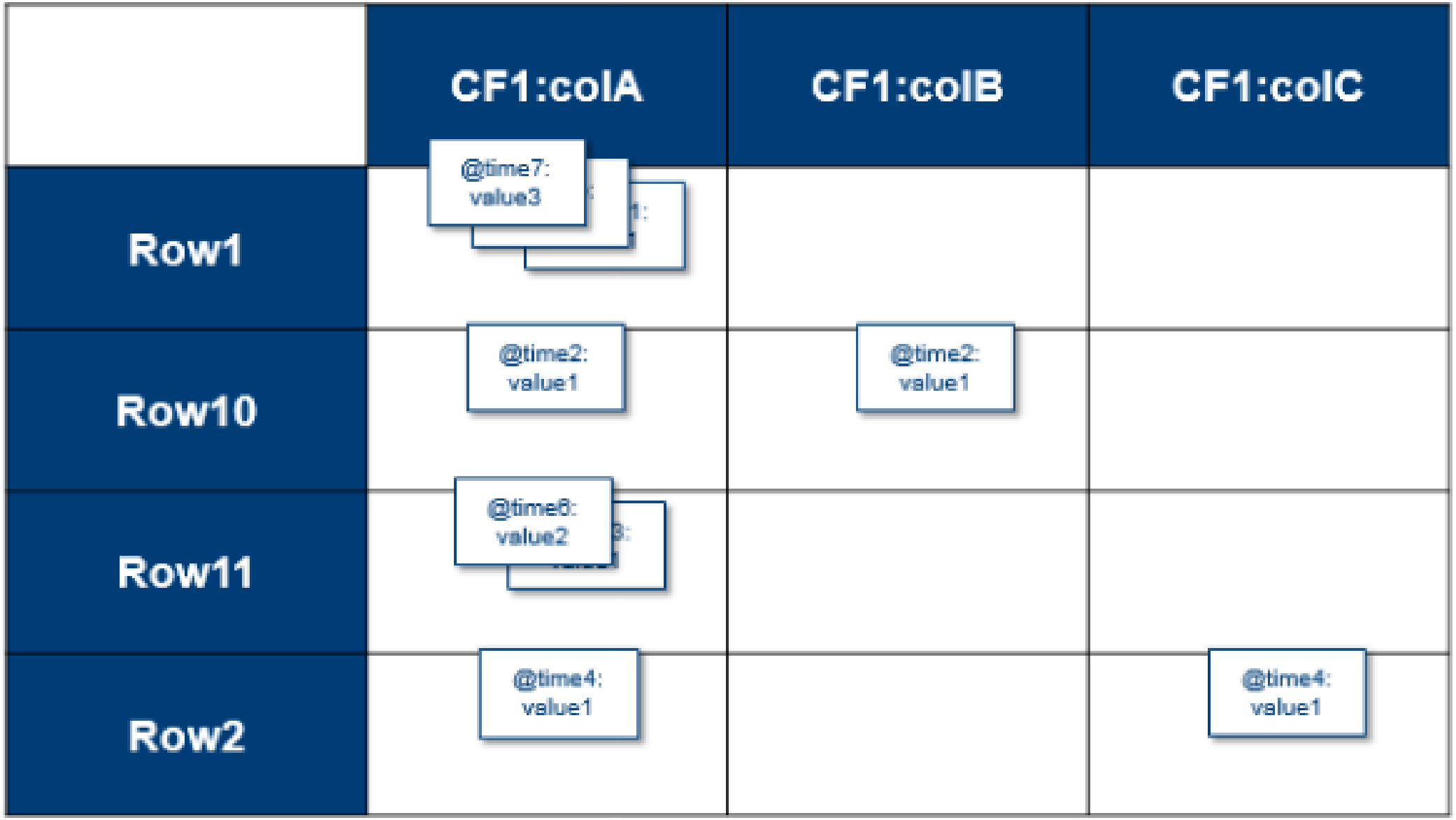
Column Families are stored in separate files

Logical data model vs. physical data storage





Sparse data with cell versions



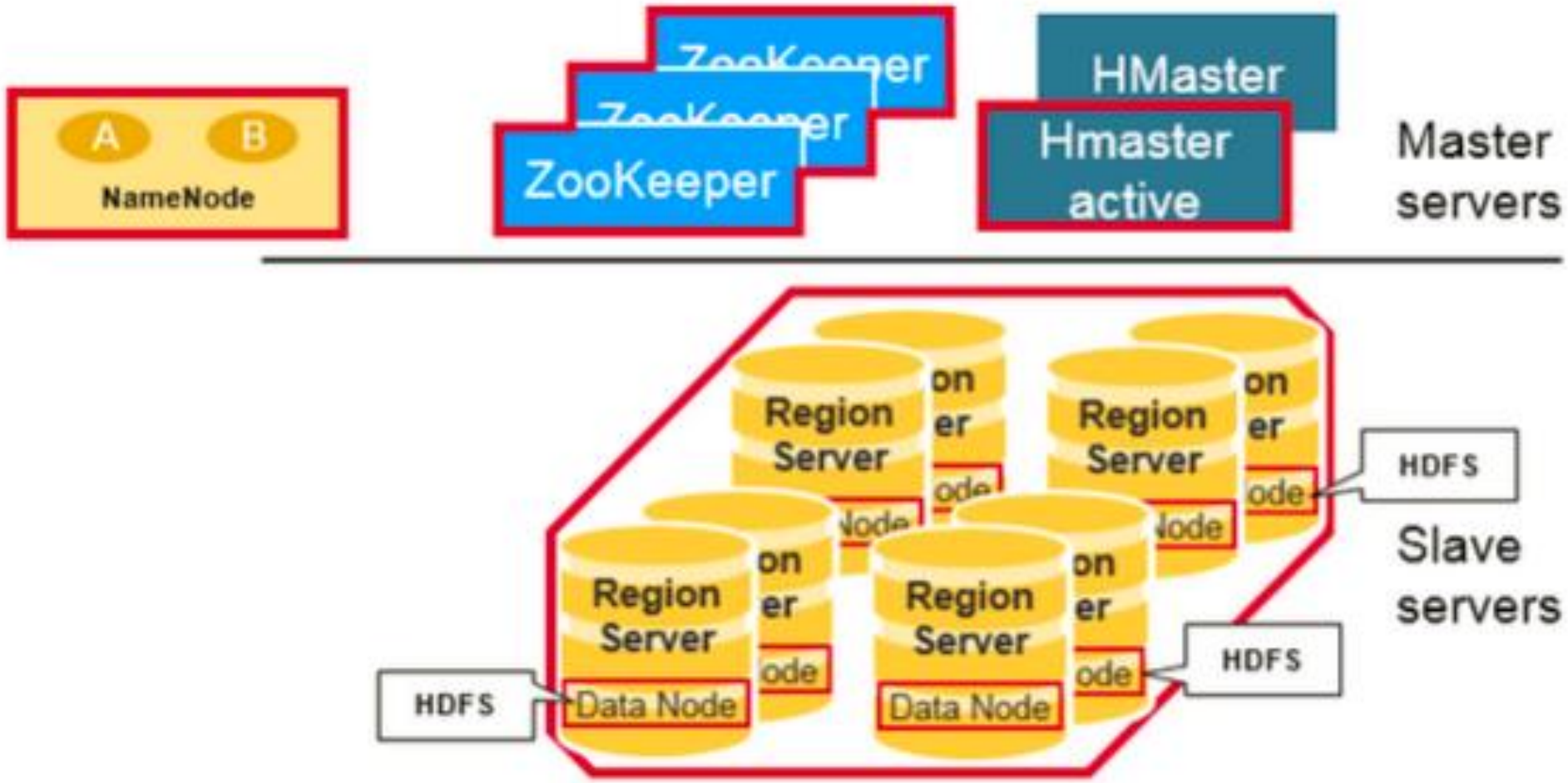
Versioned data

Number of versions can be configured. Default number equal to 1

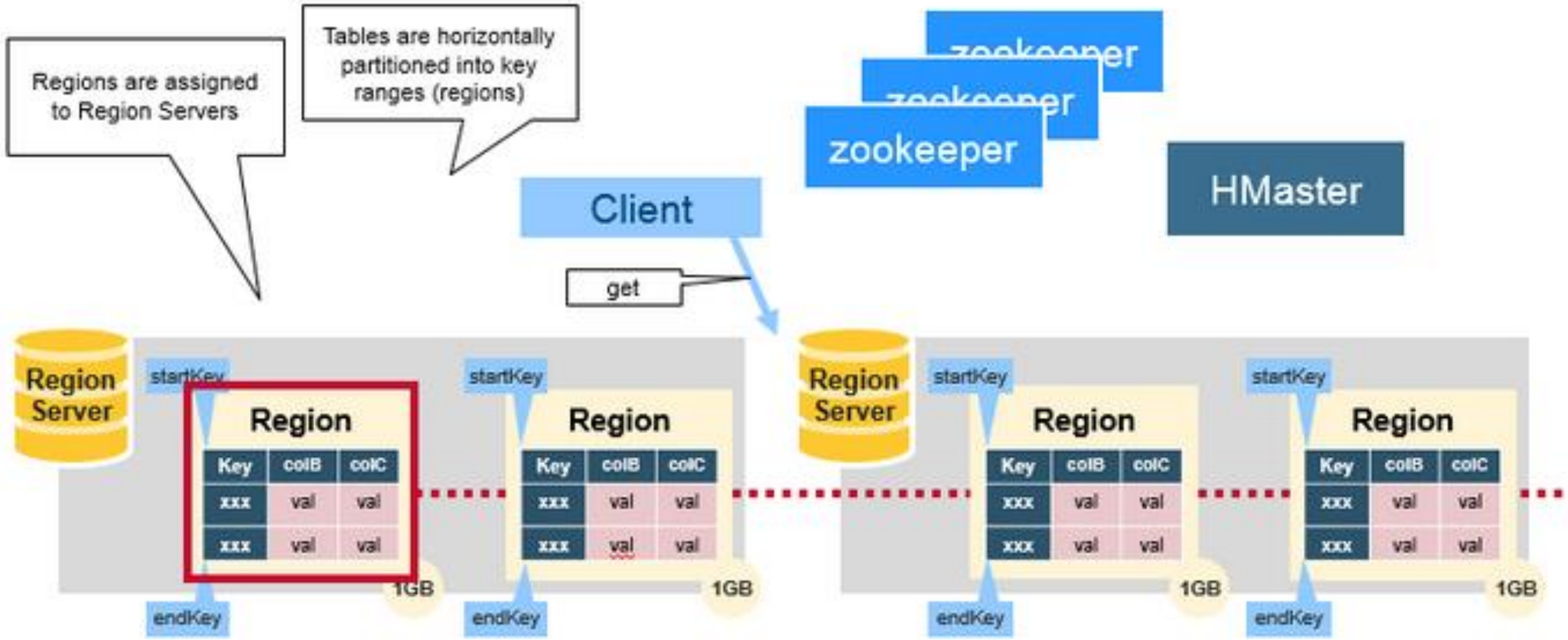
put, adds new cell

Key	CF:Col	version	value
smithj	Address:street	v3	19th Ave
smithj	Address:street	v2	Main St
smithj	Address:street	v1	Central Dr

核心架构：部署结构

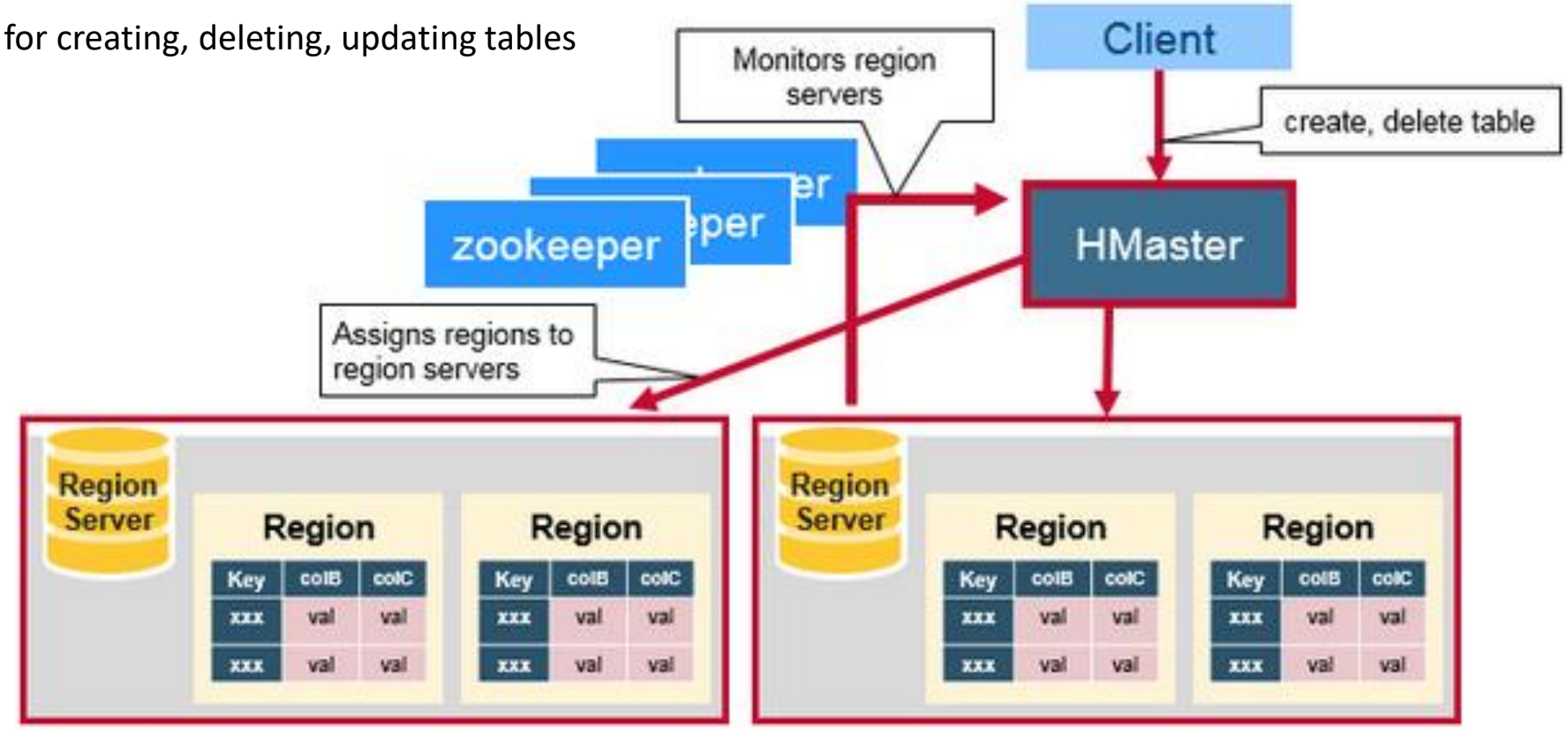

A
B
NameNode
ZooKeeper
HMaster
Hmaster
active
Master
servers
Region
Server
Data Node
HDFS
Slave
servers

Regions

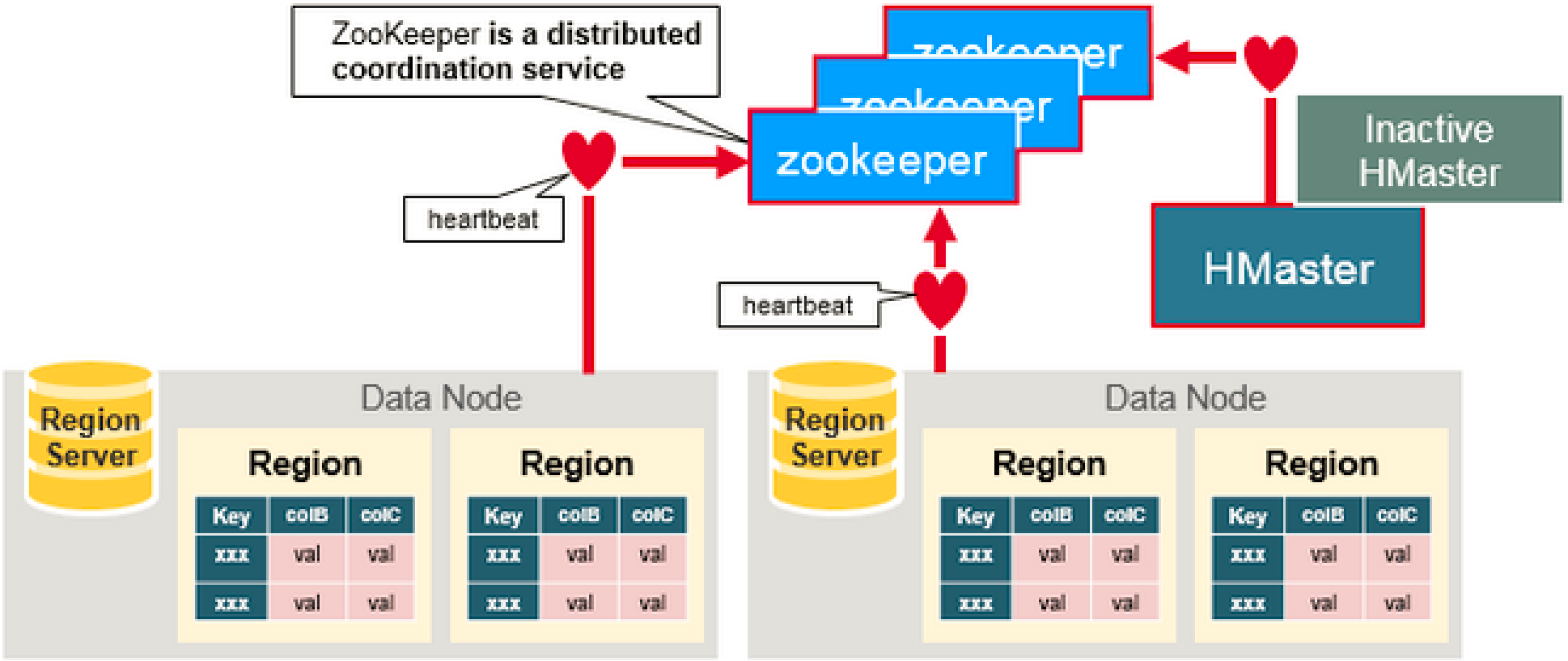

Regions are assigned to Region Servers
Tables are horizontally partitioned into key ranges (regions)
zookeeper
zookeeper
zookeeper
HMaster
Client
get
Region Server
startKey
Region
Key
colB
colC
xxx
val
val
xxx
val
val
endKey
1GB
startKey
Region
Key
colB
colC
xxx
val
val
xxx
val
val
endKey
1GB
Region Server
startKey
Region
Key
colB
colC
xxx
val
val
xxx
val
val
endKey
1GB
startKey
Region
Key
colB
colC
xxx
val
val
xxx
val
val
endKey
1GB

HBase HMaster

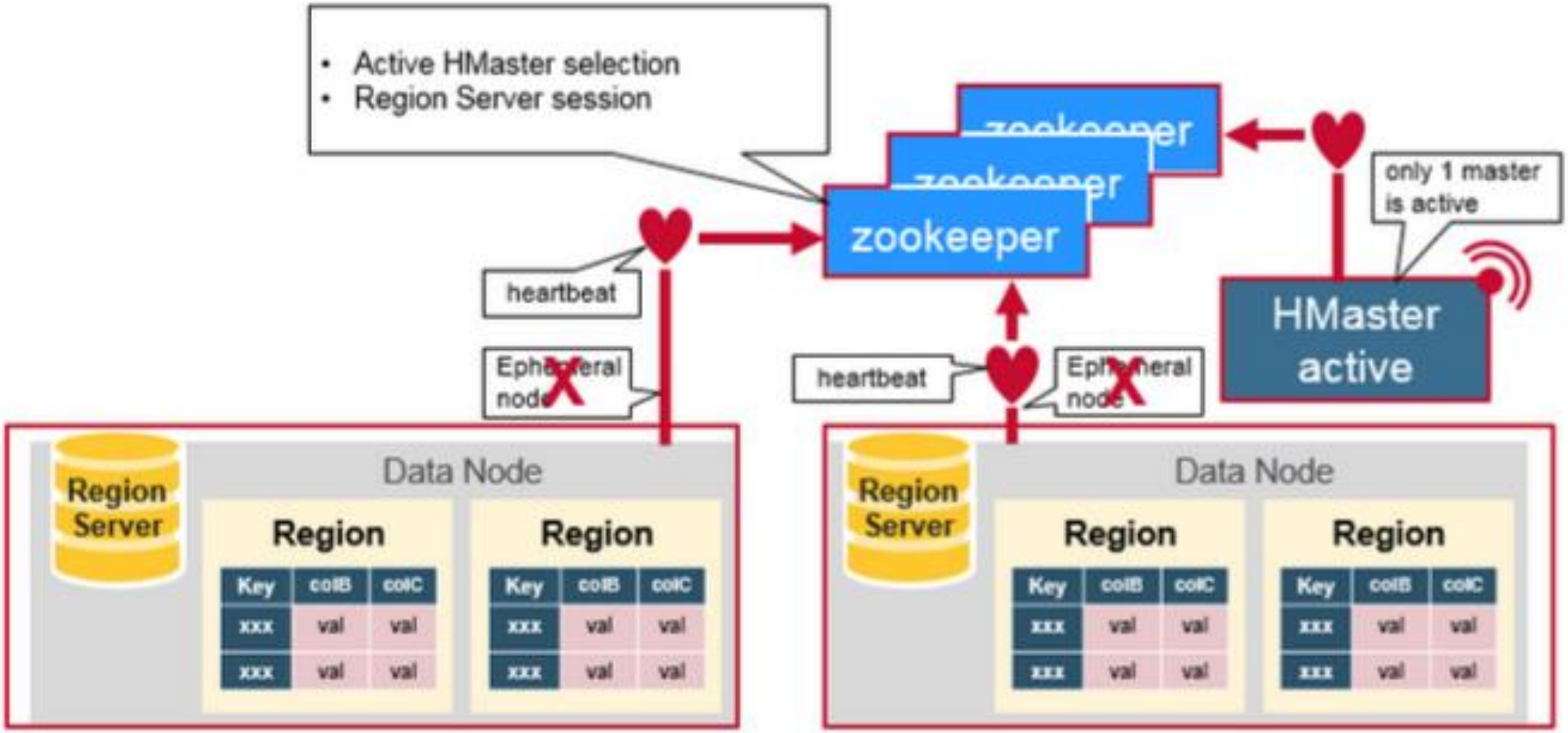
- **Coordinating the region servers**
  - - Assigning regions on startup , re-assigning regions for recovery or load balancing
  - - Monitoring all RegionServer instances in the cluster (listens for notifications from zookeeper)
- **Admin functions**
  - - Interface for creating, deleting, updating tables



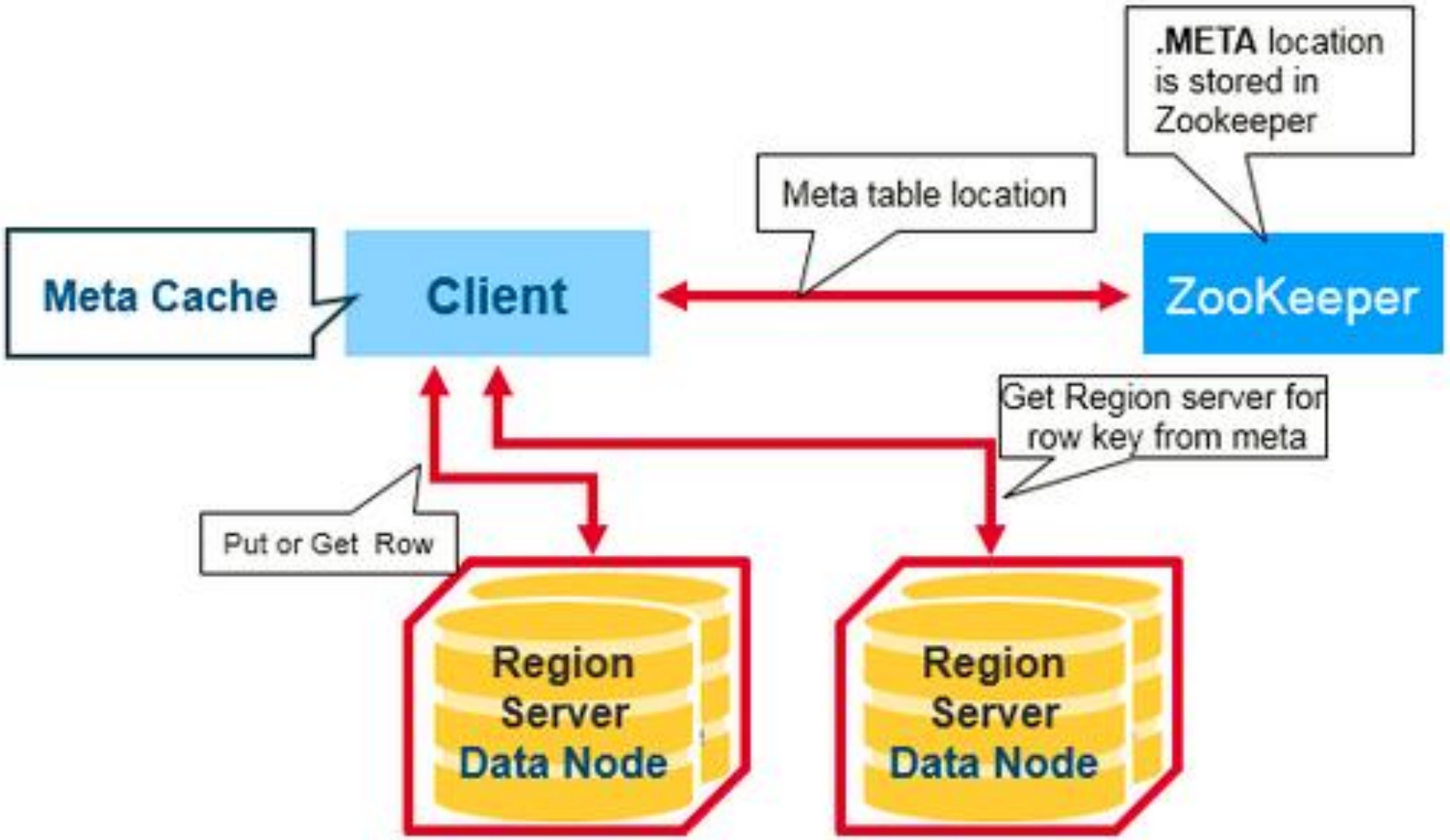
ZooKeeper: The Coordinator



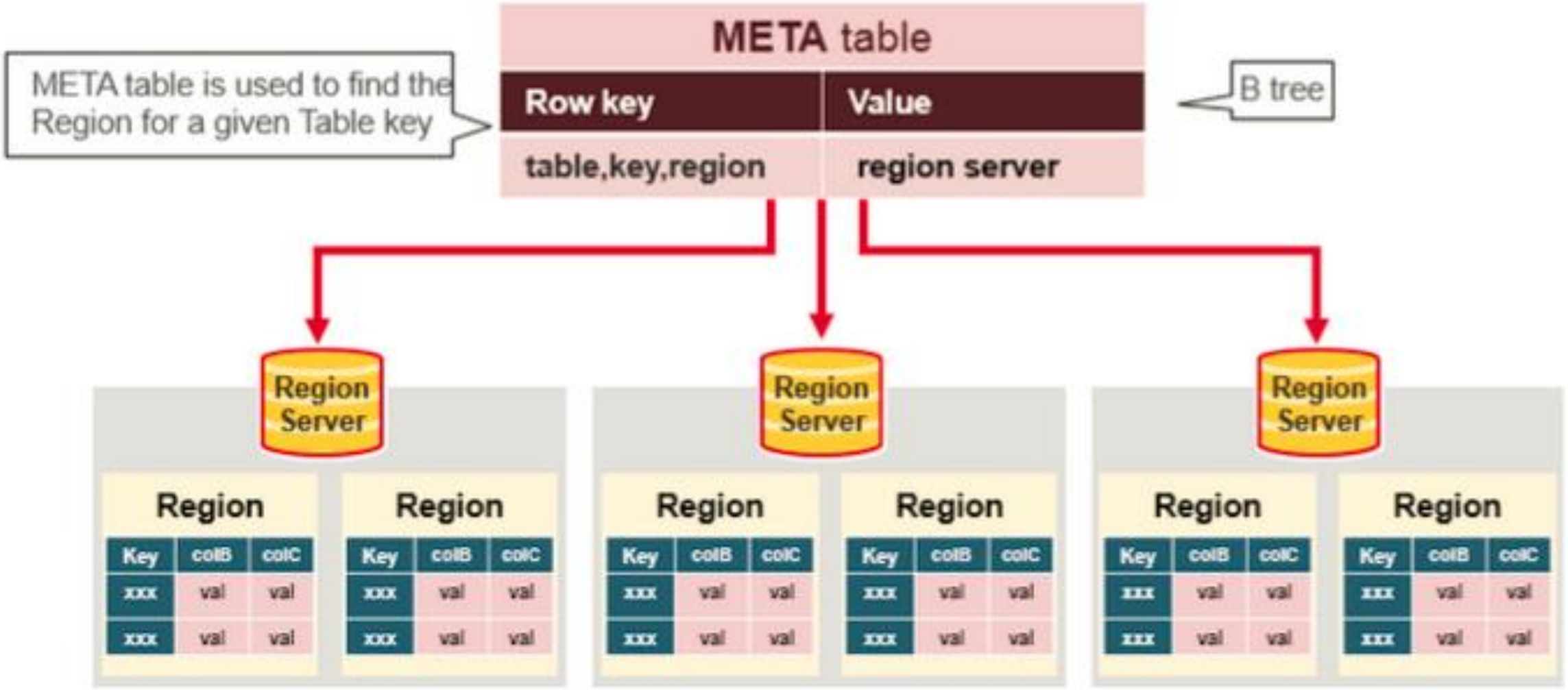
How the Components Work Together



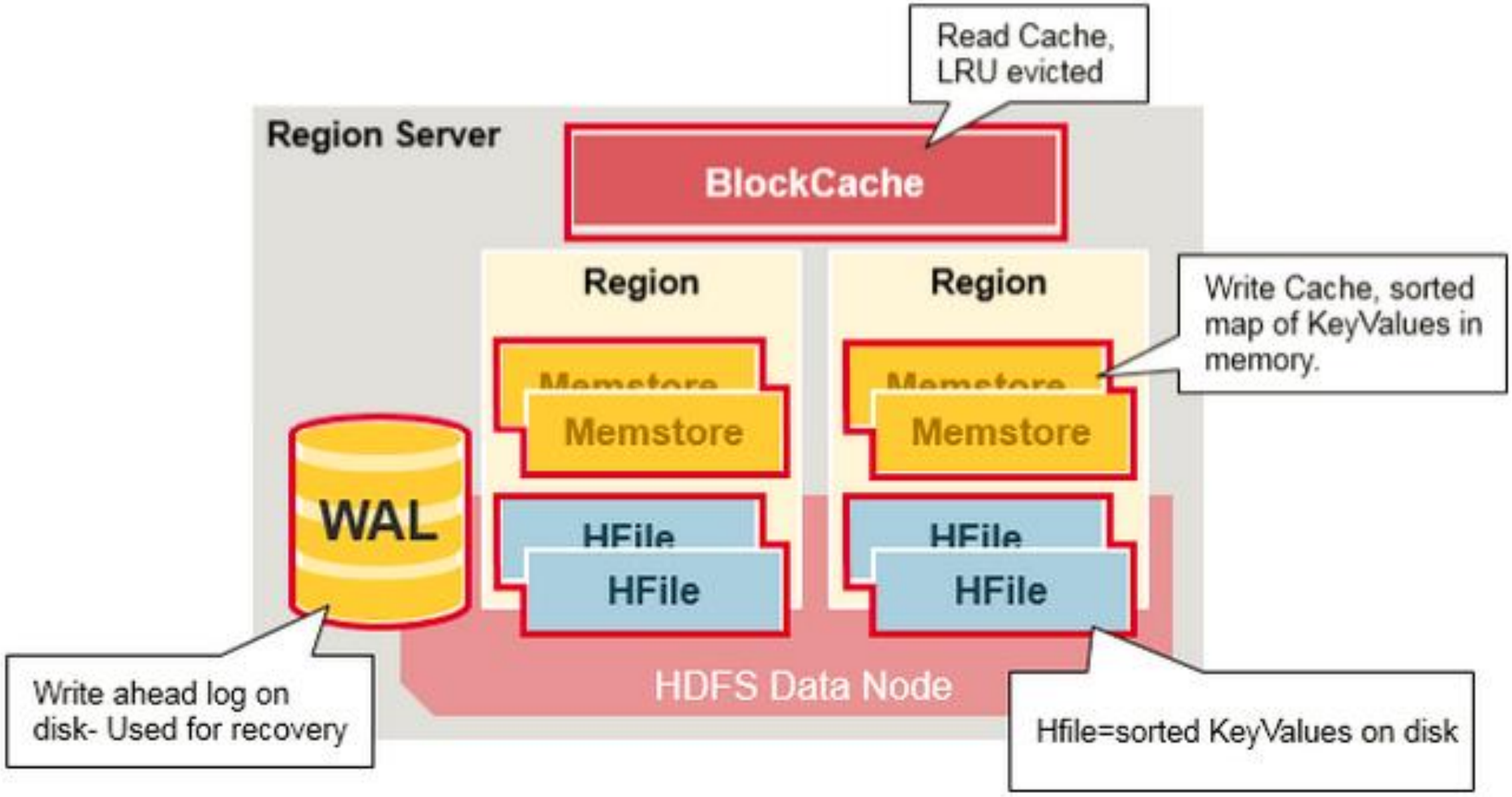
HBase First Read or Write



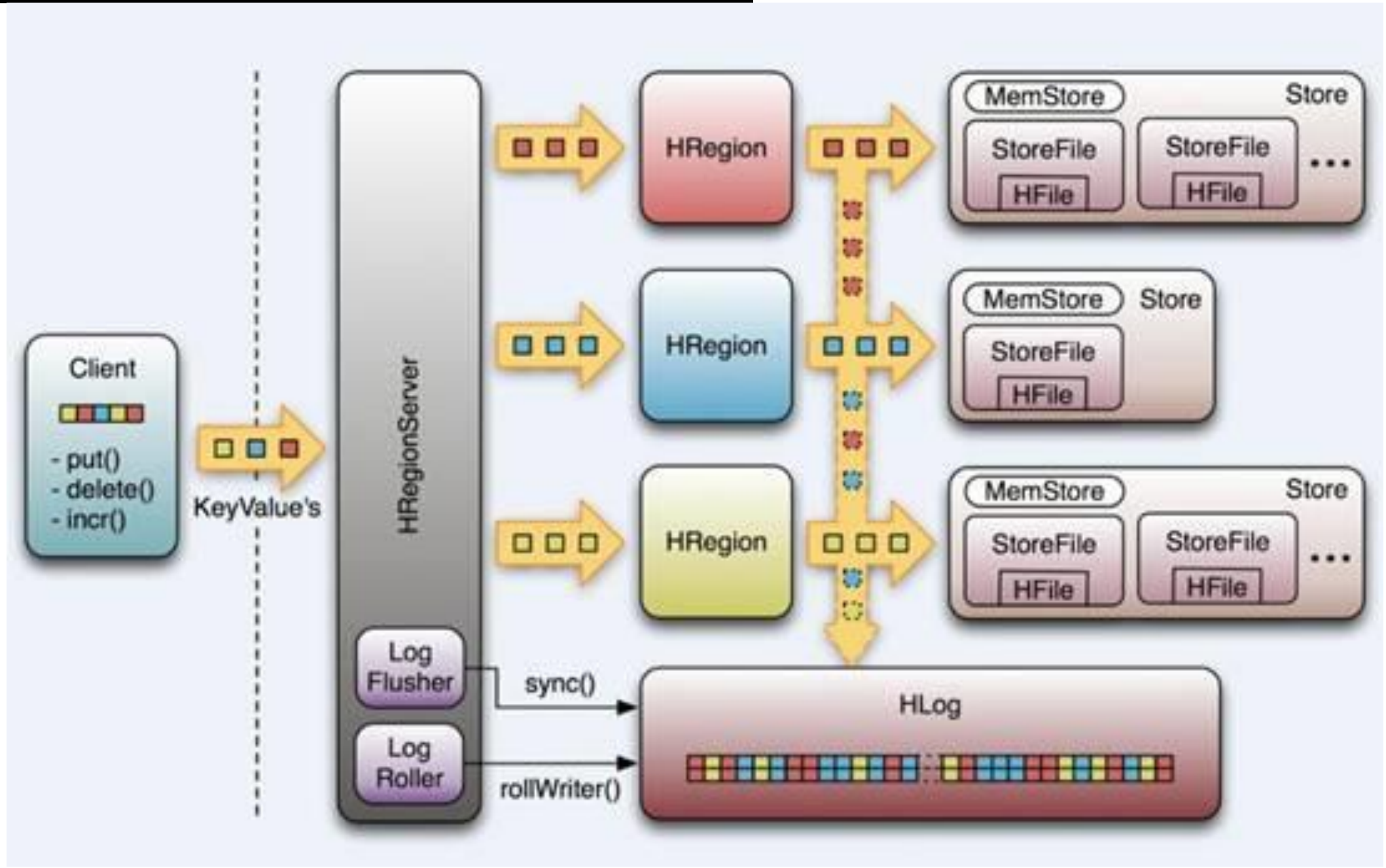
HBase Meta Table



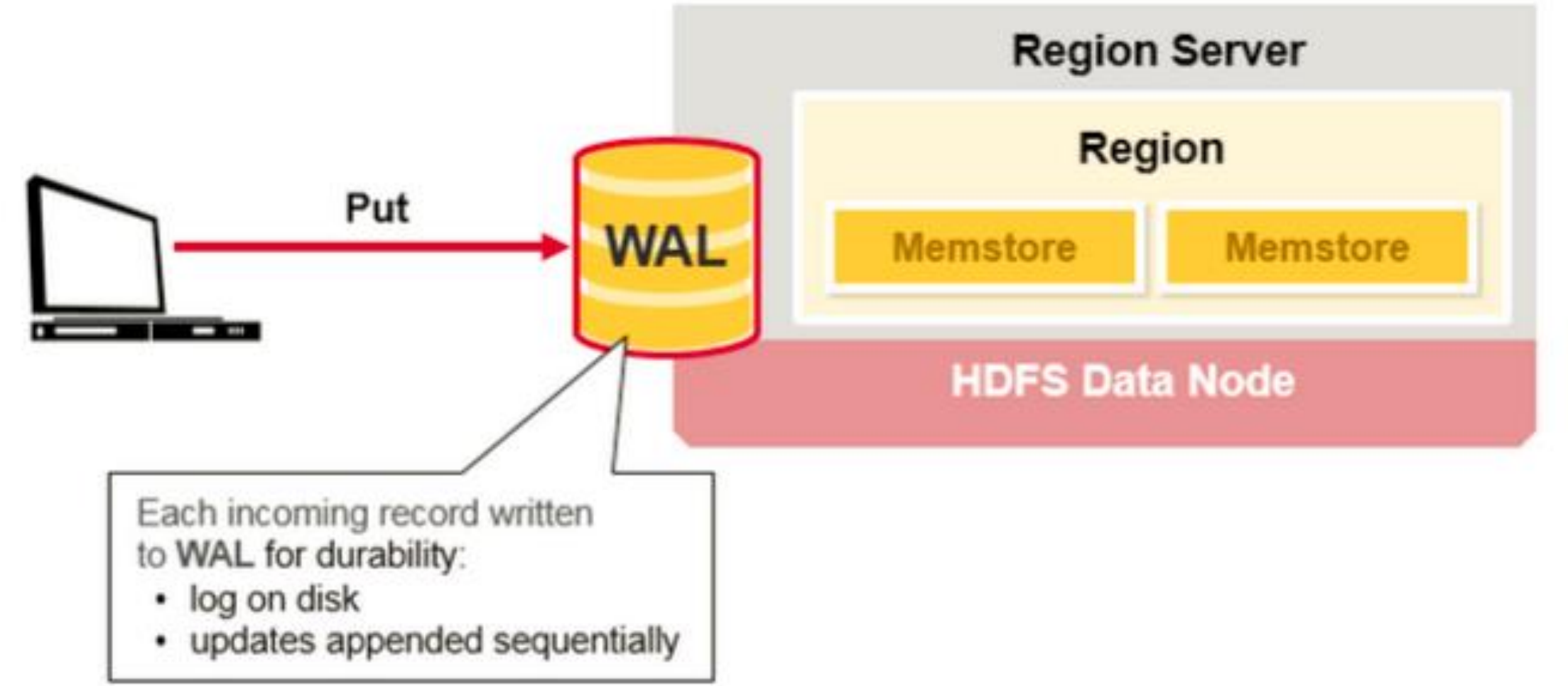
Region Server Components



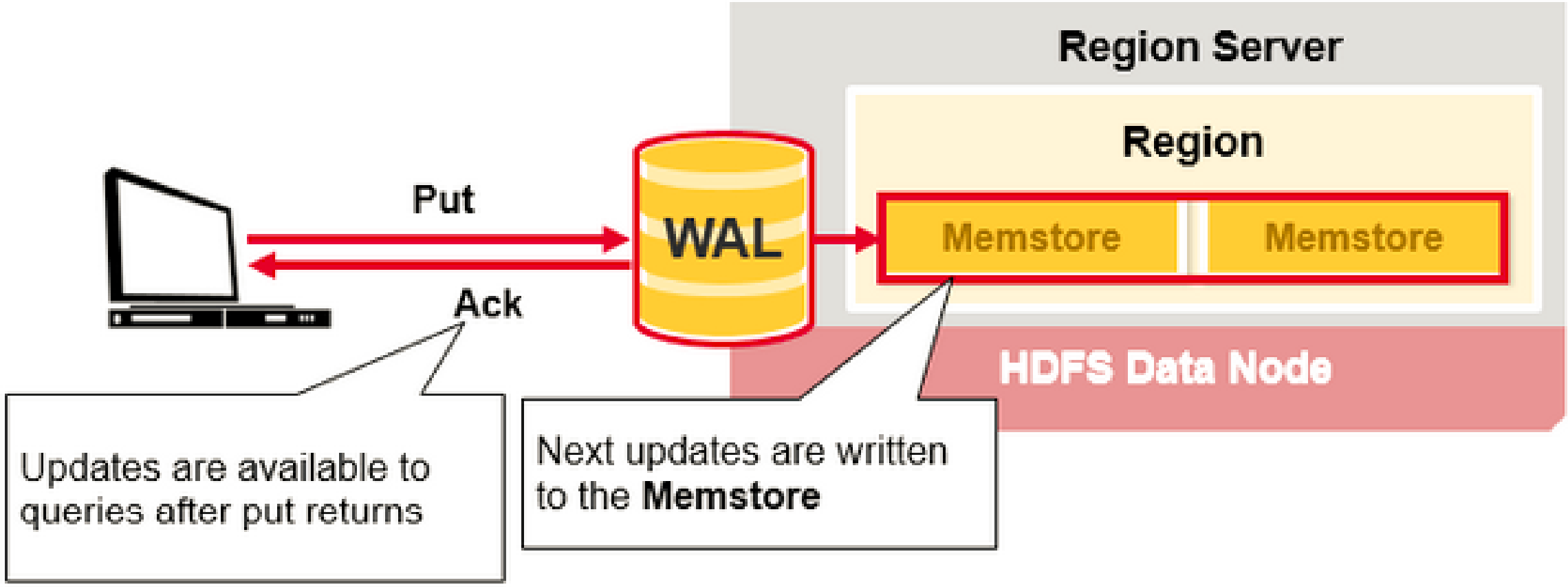
Region Server Components



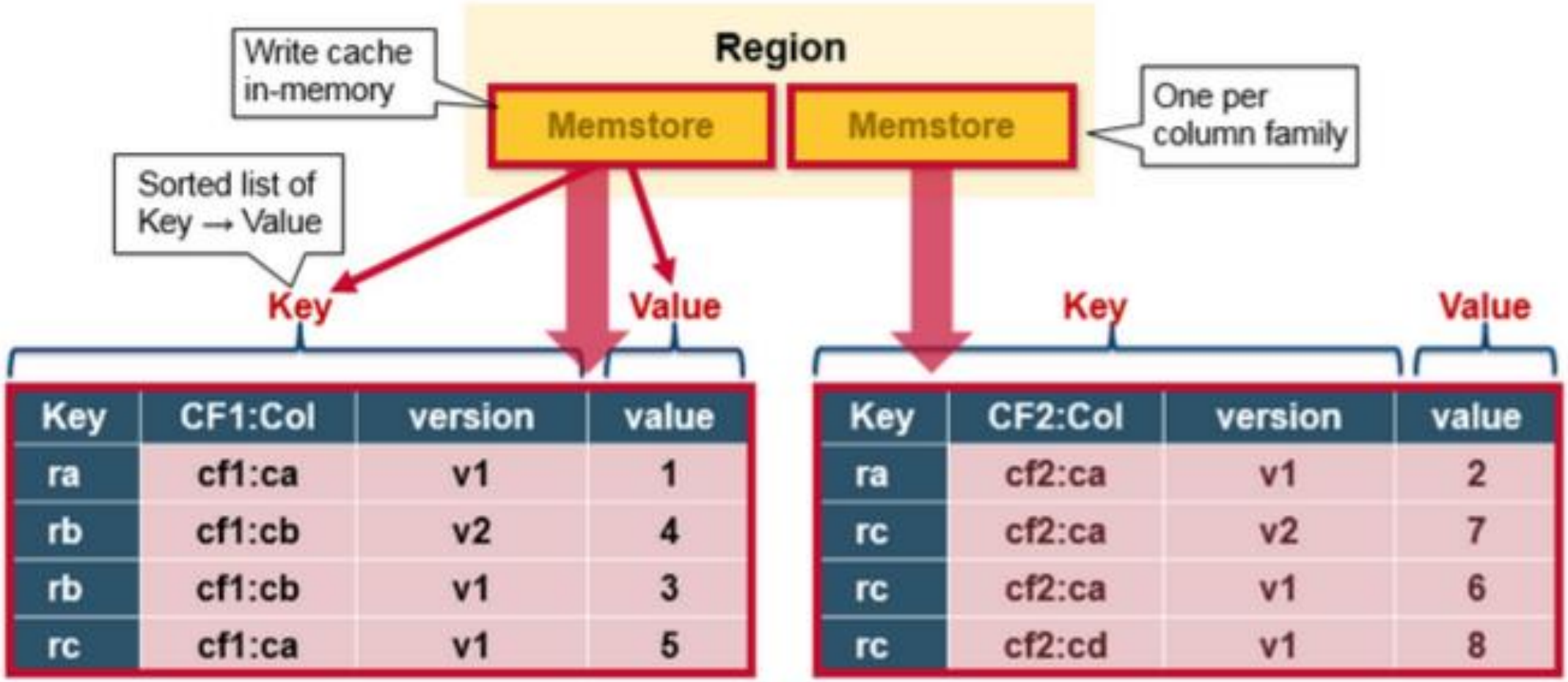
HBase Write Steps (1)



HBase Write Steps (2)



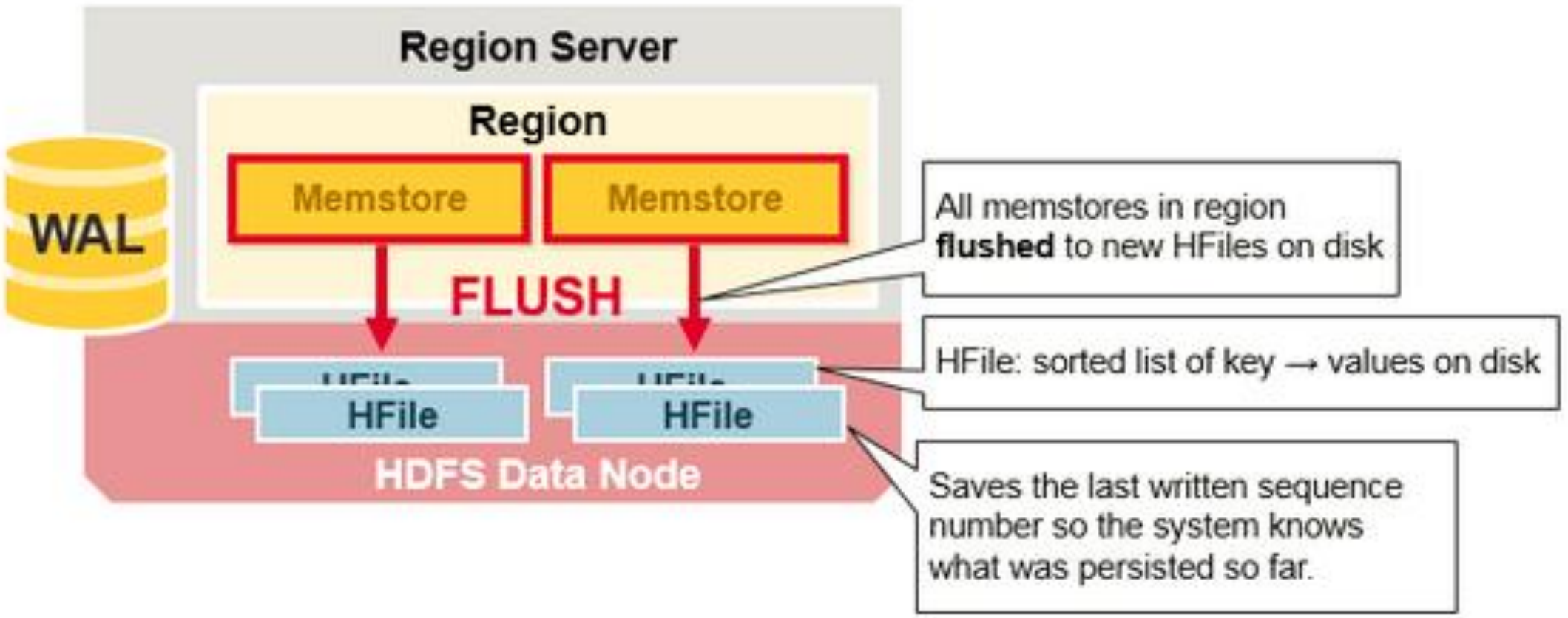
HBase MemStore



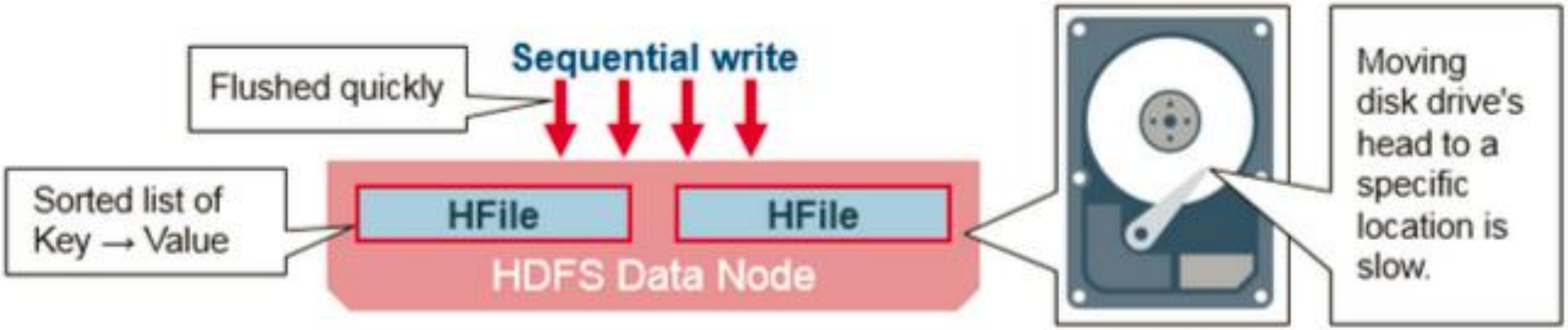
Key	CF1:Col	version	value
ra	cf1:ca	v1	1
rb	cf1:cb	v2	4
rb	cf1:cb	v1	3
rc	cf1:ca	v1	5

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 Region Flush



HBase HFile



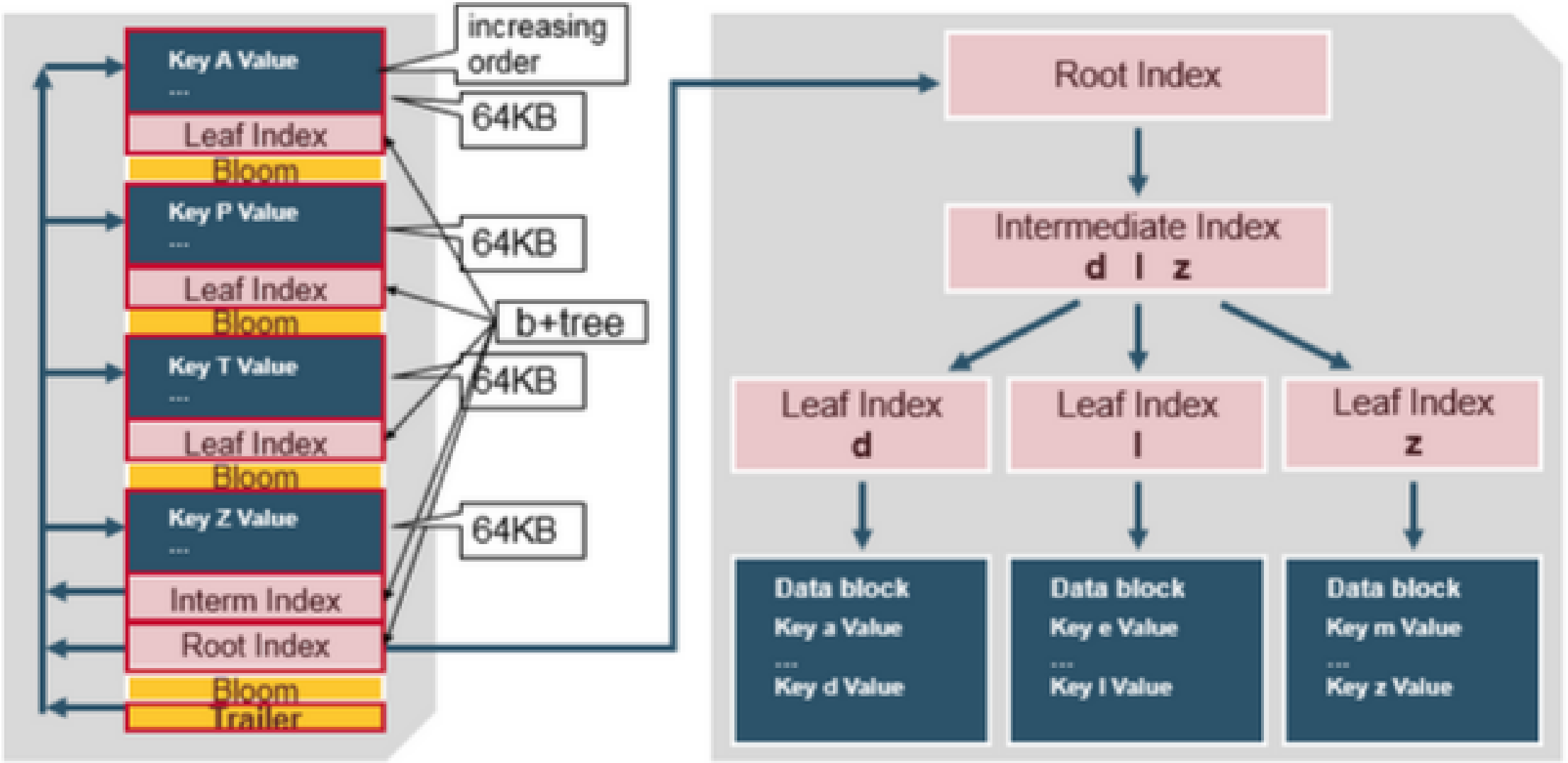
Key Value

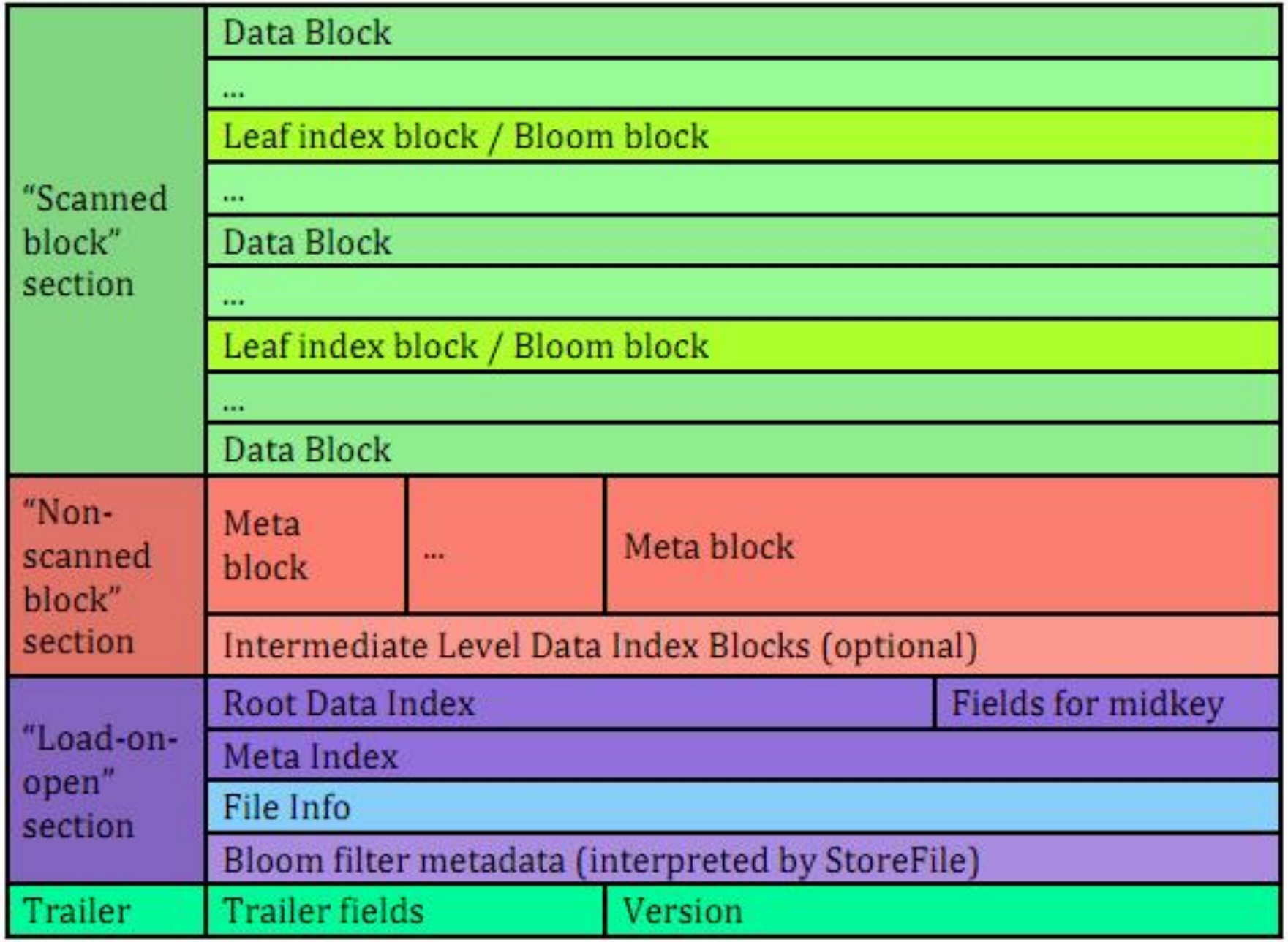
Key	CF1:Col	version	value
ra	cf1:ca	v1	1
rb	cf1:cb	v2	4
rb	cf1:cb	v1	3
rc	cf1:ca	v1	5

Key Value

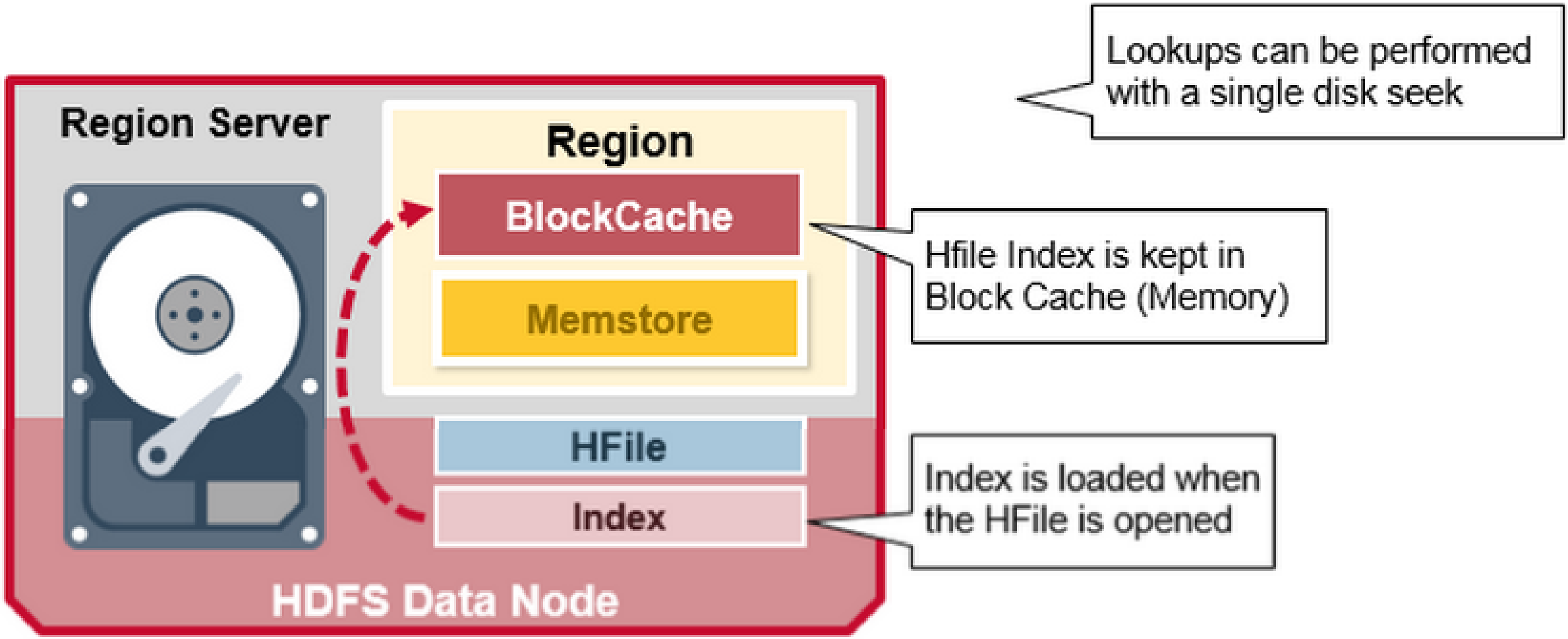
Key	CF2:Col	version	value
ra	cf2:ca	v1	2
rc	cf2:ca	v2	7
rc	cf2:ca	v1	6
rc	cf2:cd	v1	8

HBase HFile Structure



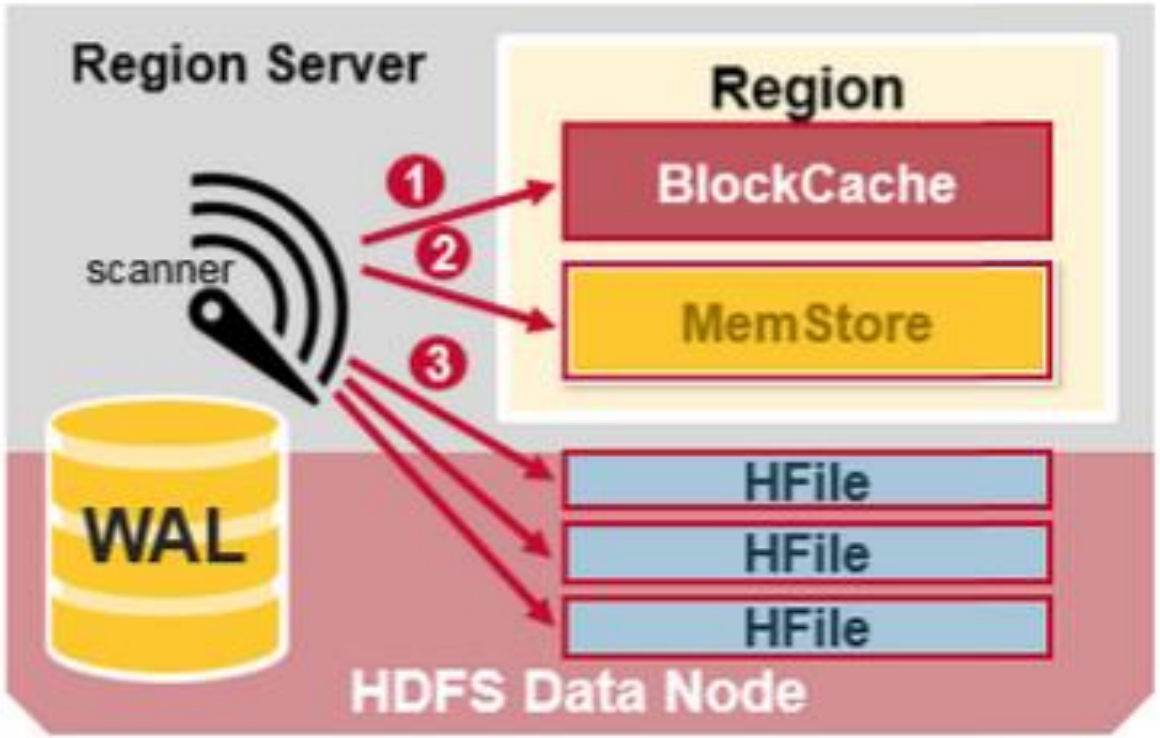

"Scanned block" section
Data Block
...
Leaf index block / Bloom block
...
Data Block
...
Leaf index block / Bloom block
...
Data Block
"Non-scanned block" section
Meta block
...
Meta block
Intermediate Level Data Index Blocks (optional)
"Load-on-open" section
Root Data Index
Fields for midkey
Meta Index
File Info
Bloom filter metadata (interpreted by StoreFile)
Trailer
Trailer fields
Version

HFile Index

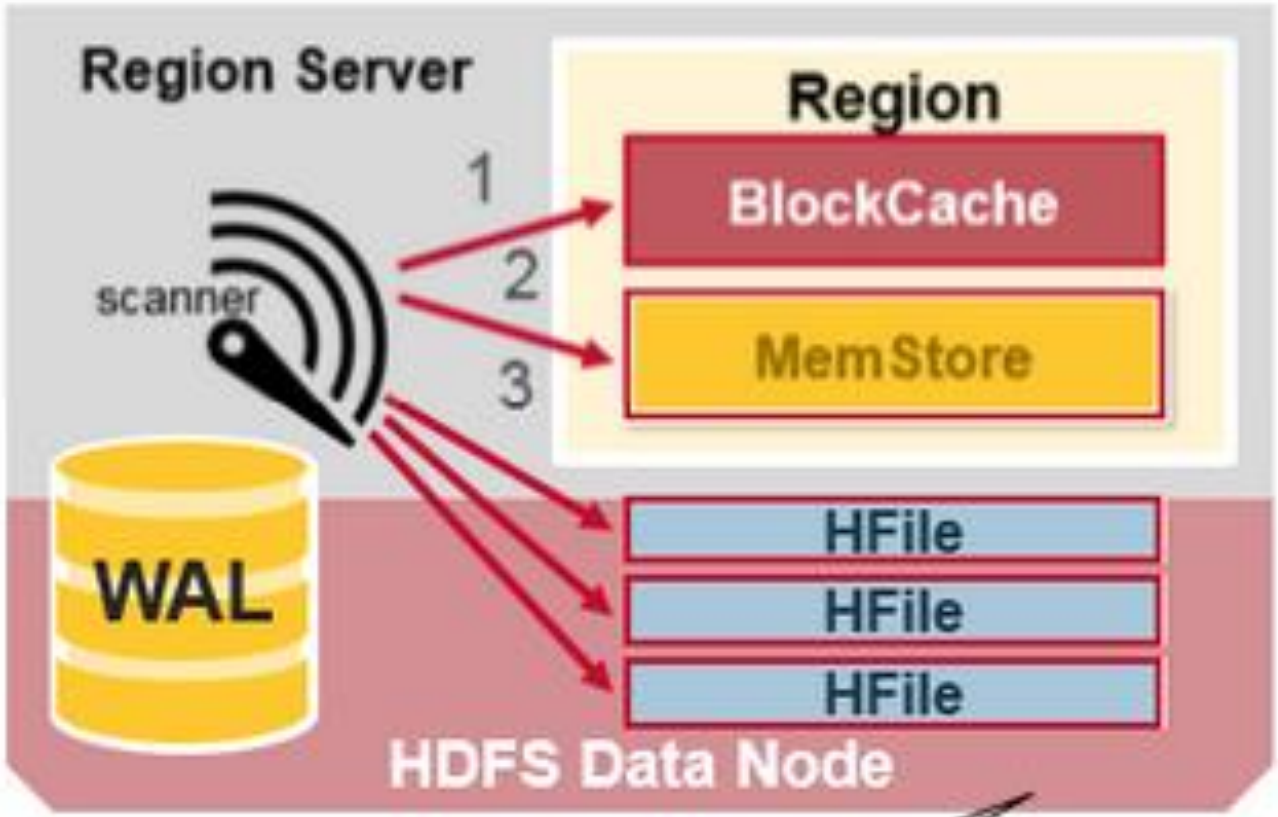

Region Server
Region
BlockCache
Memstore
HFile
Index
HDFS Data Node
Lookups can be performed with a single disk seek
Hfile Index is kept in Block Cache (Memory)
Index is loaded when the HFile is opened

HBase Read Merge

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



HBase Read Merge

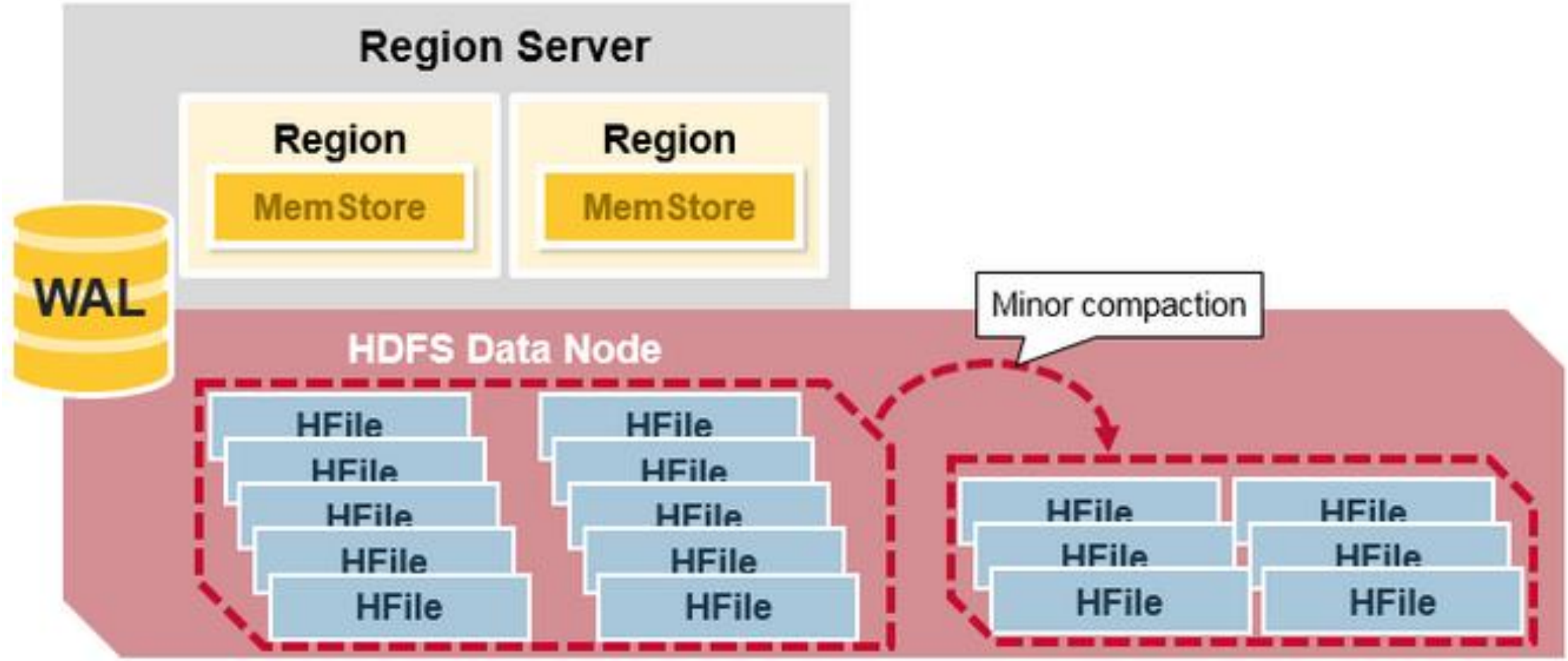


MemStore creates multiple **small store files** over time when **flushing.**

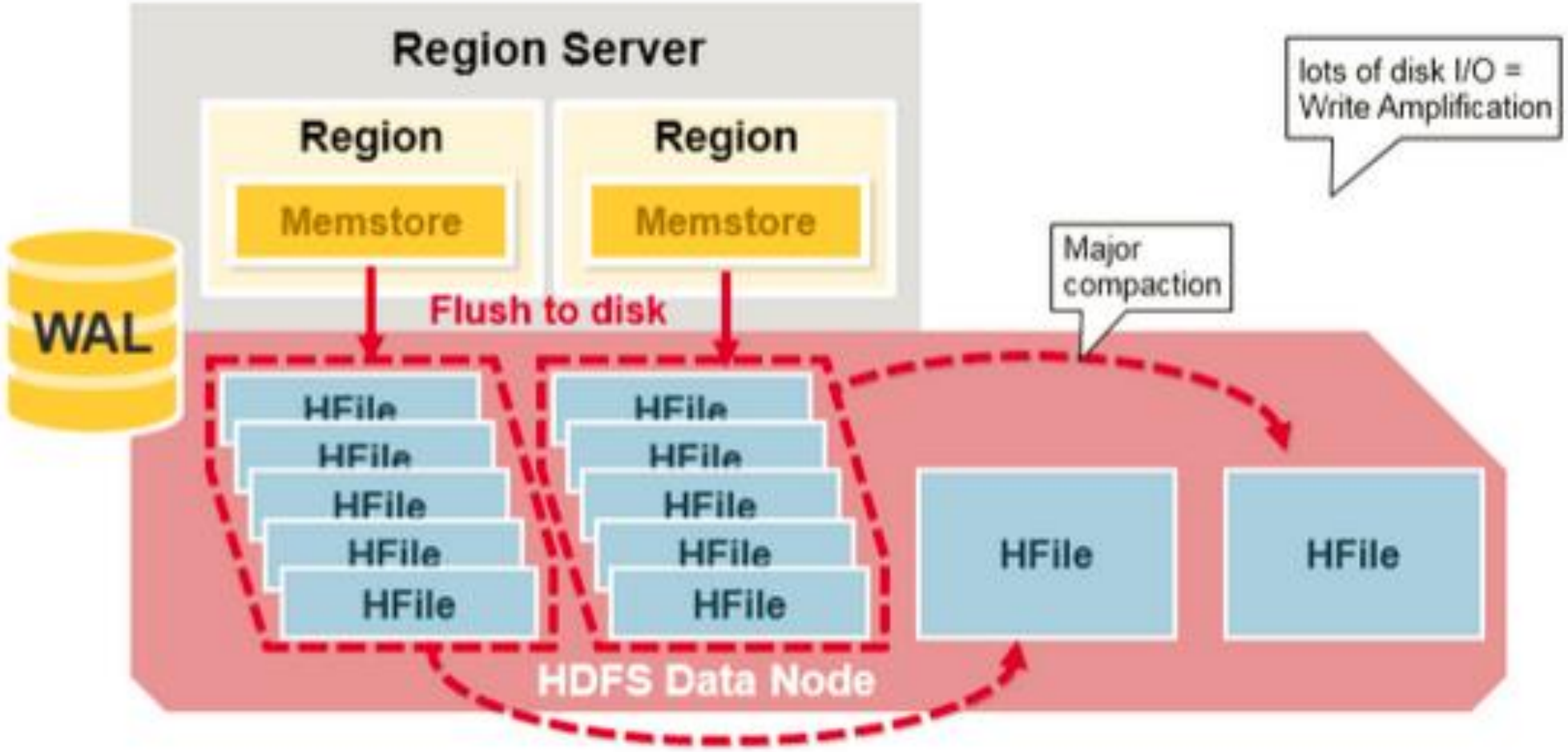
Read Amplification

- multiple files have to be examined

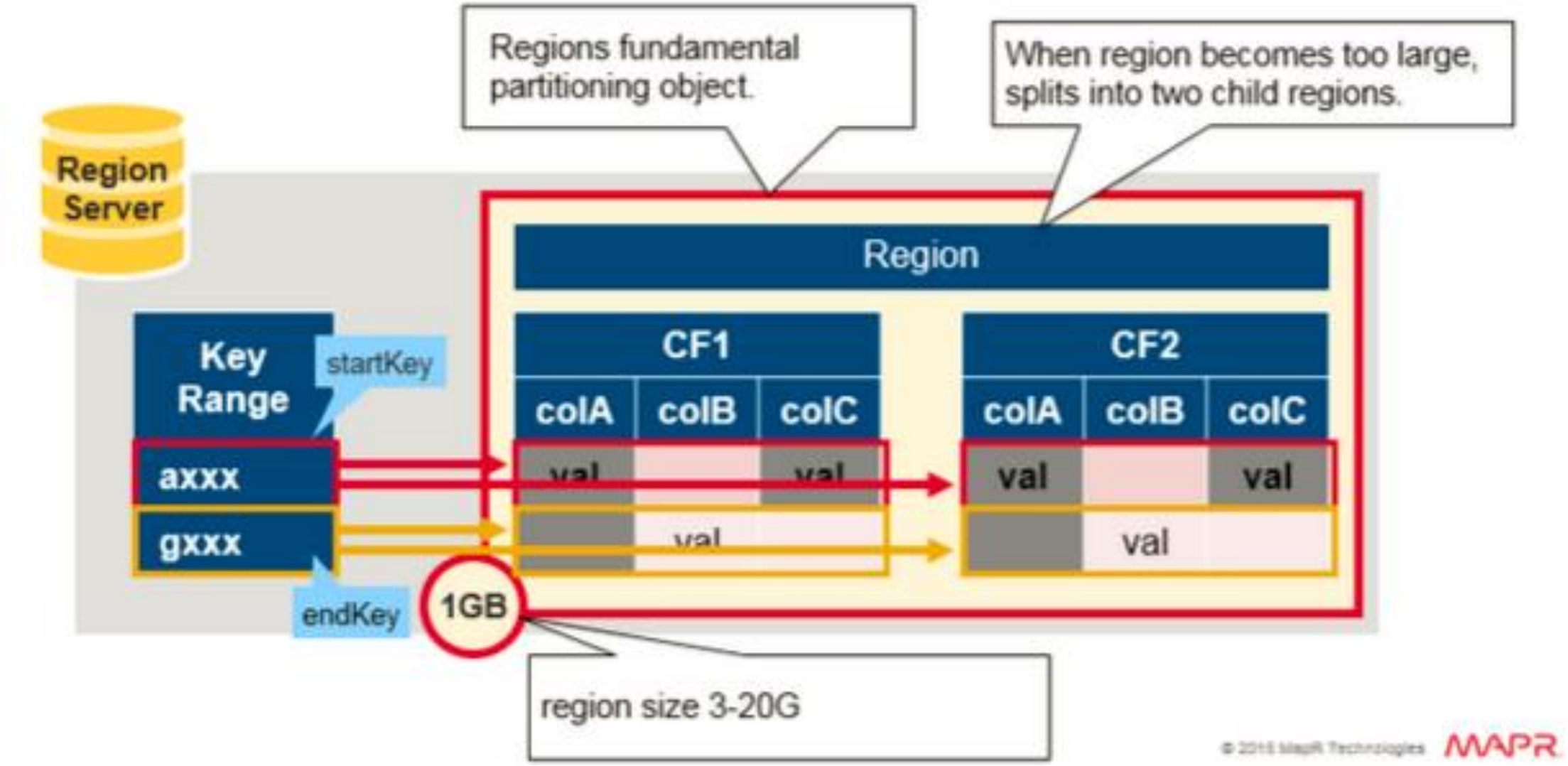
HBase Minor Compaction



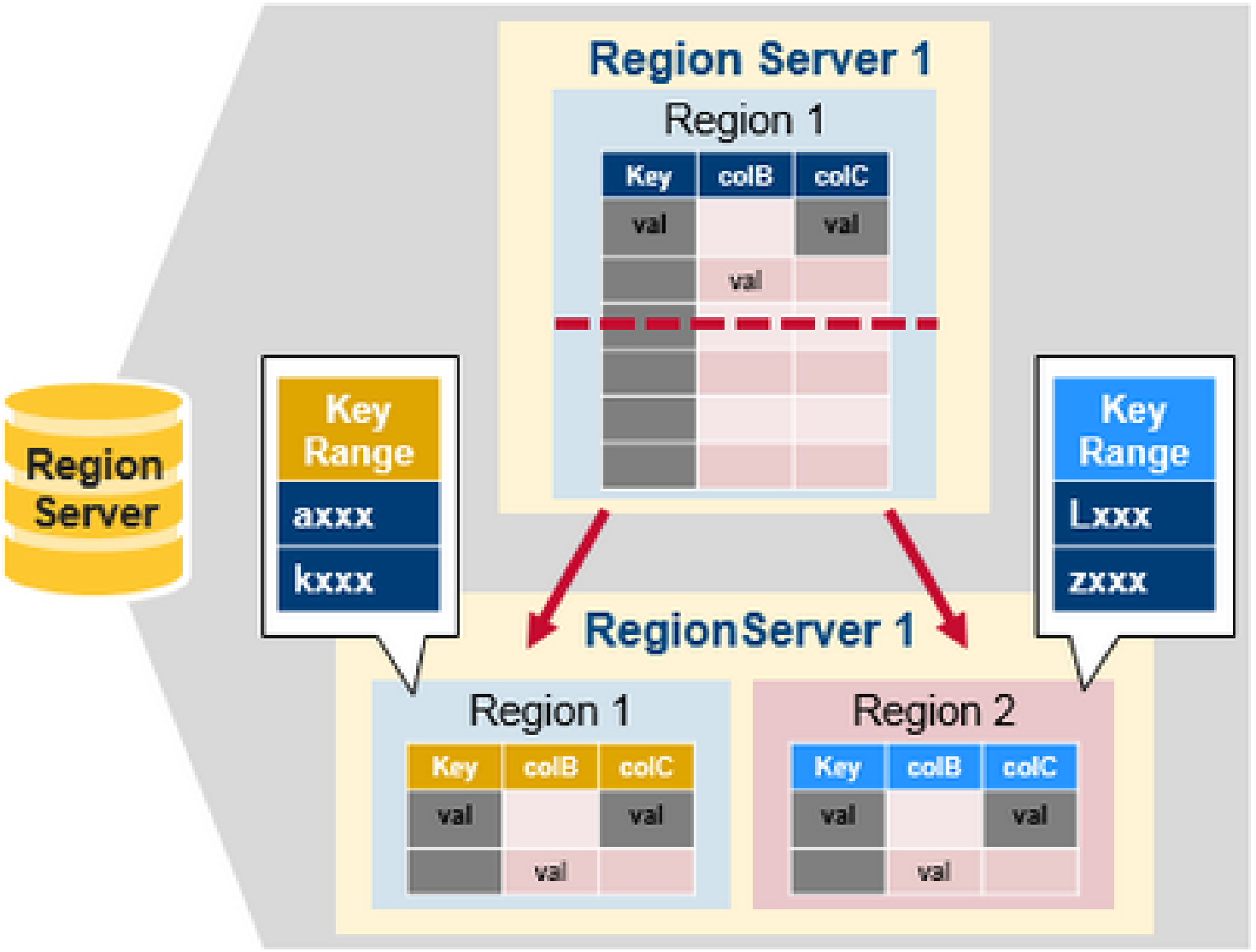
HBase Major Compaction



Region = Contiguous Keys

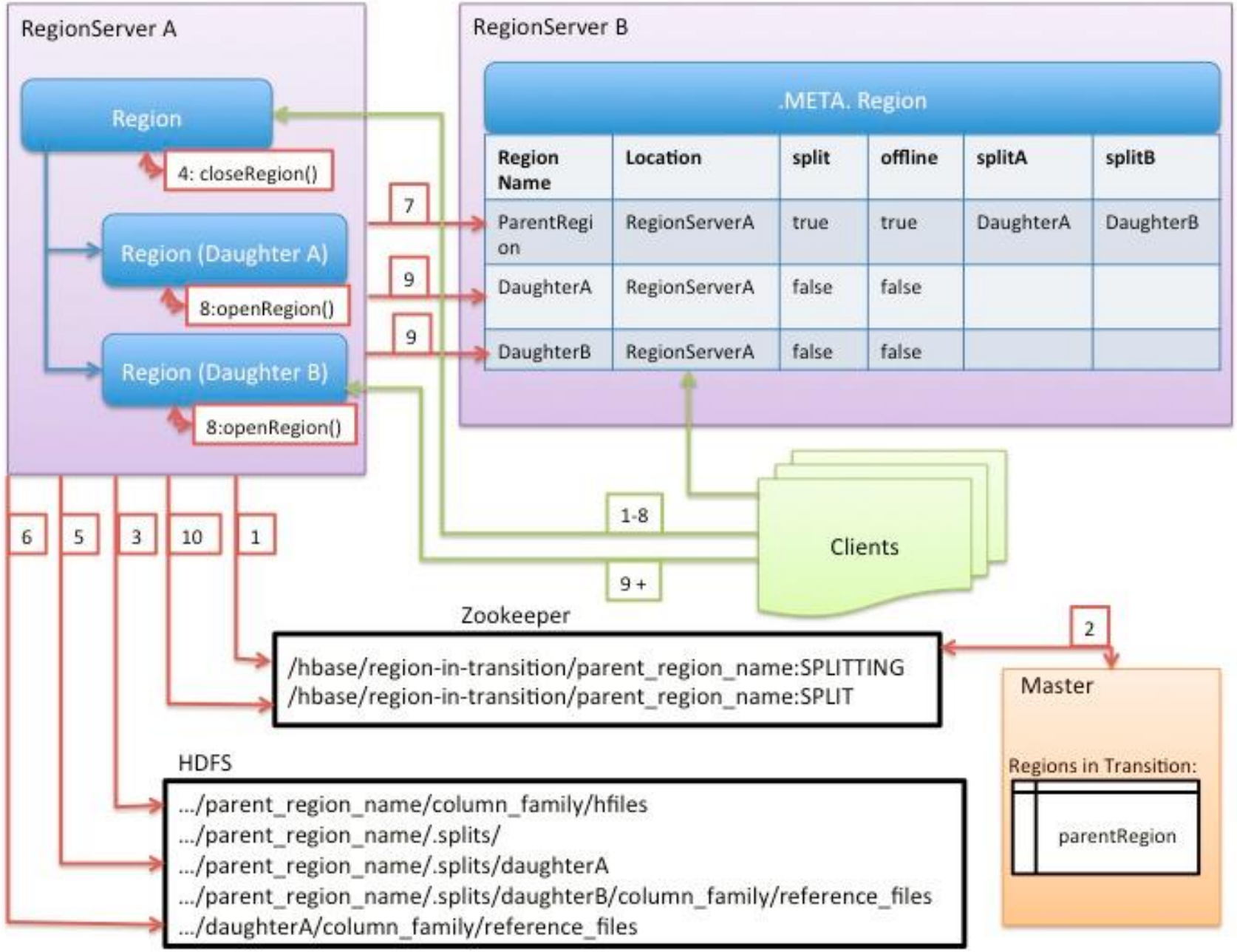


Region Split

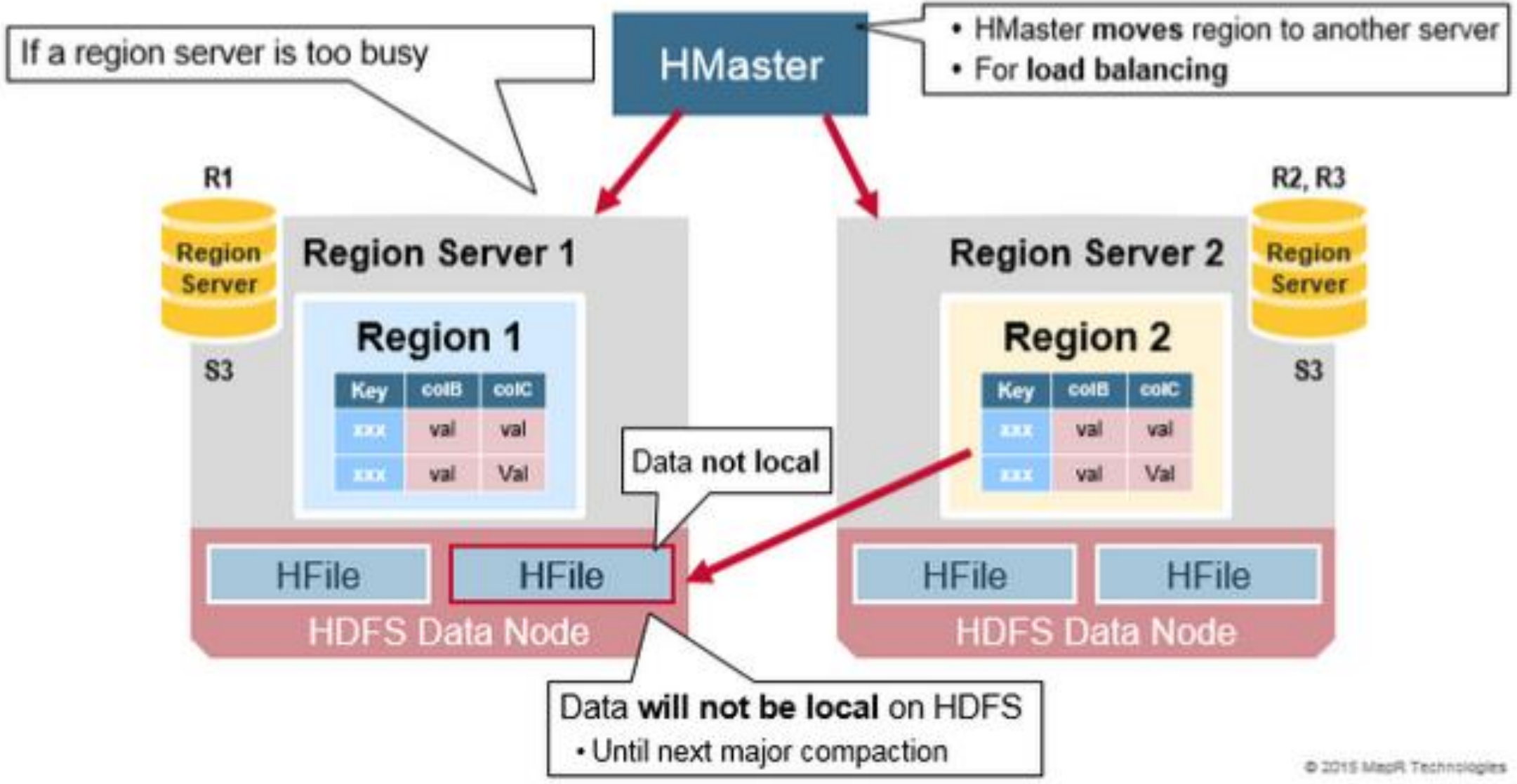


when region size > hbase.hregion.max.filesize → split

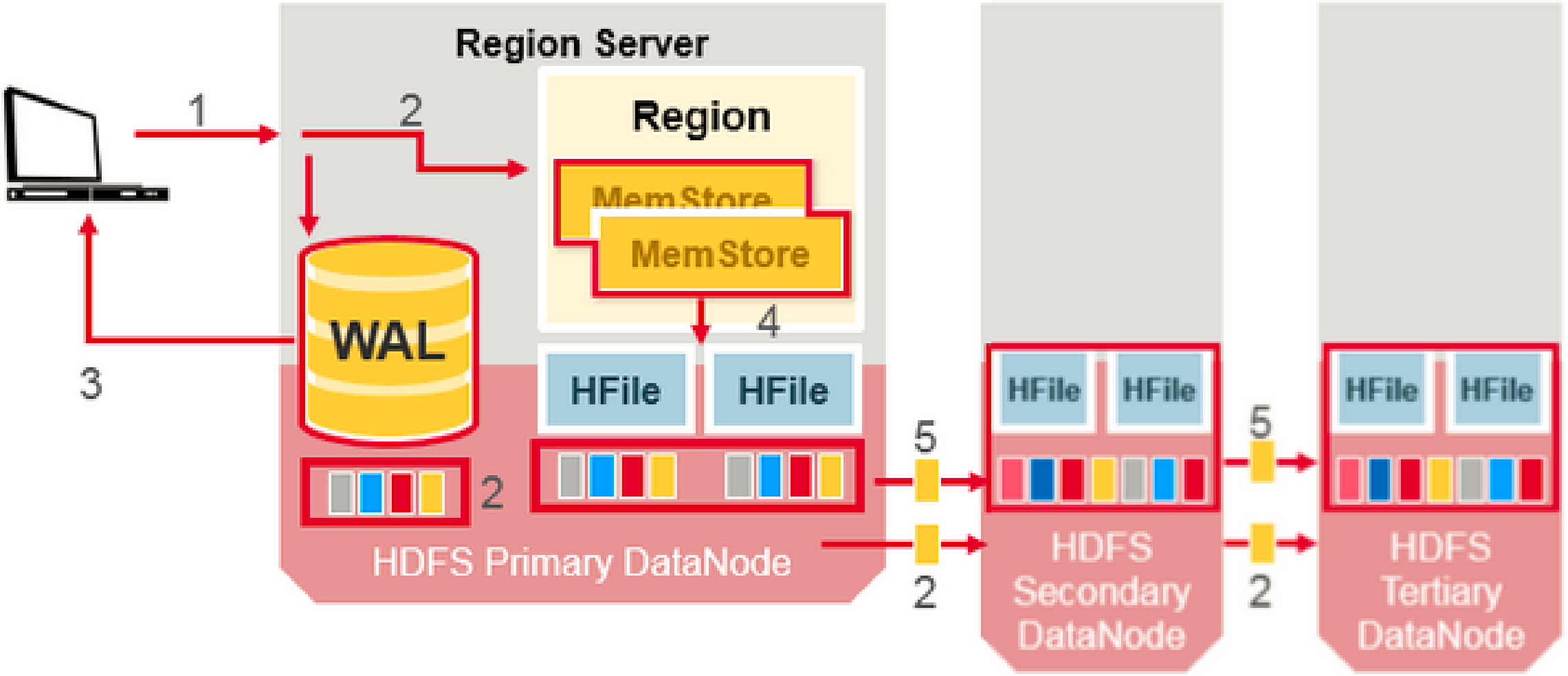
HOW REGION SPLITS ARE IMPLEMENTED



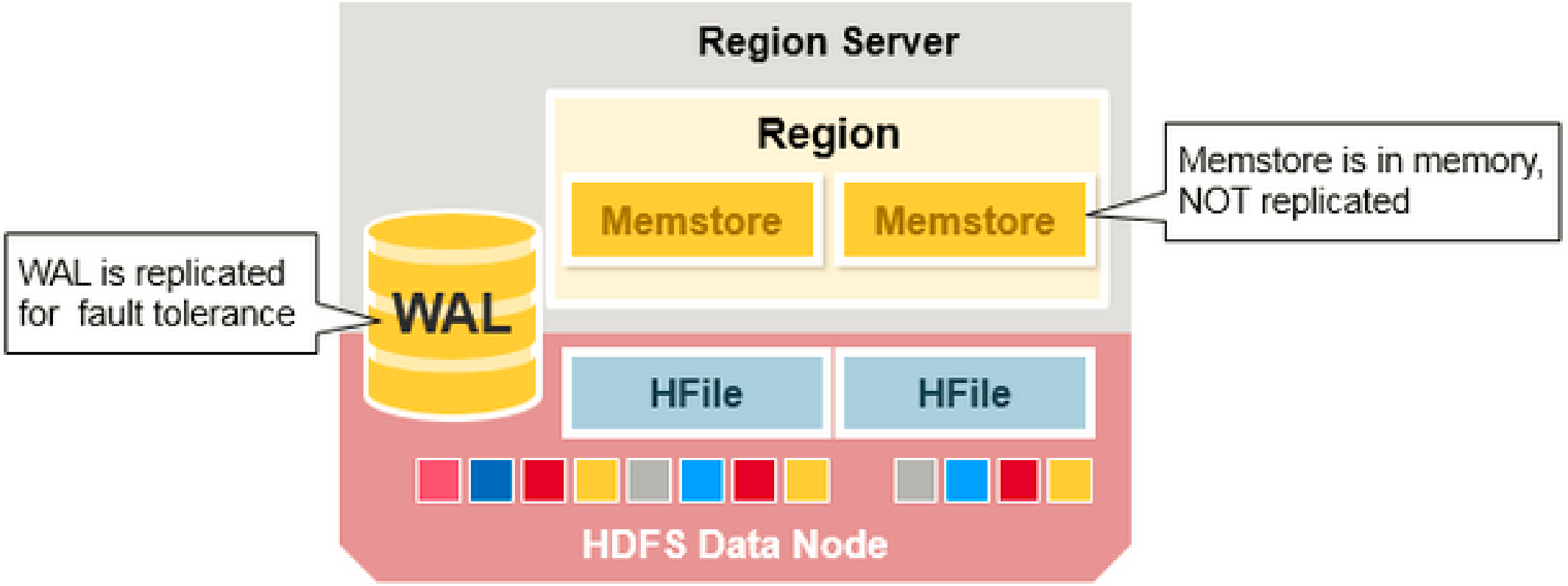
Read Load Balancing



HDFS Data Replication(1)

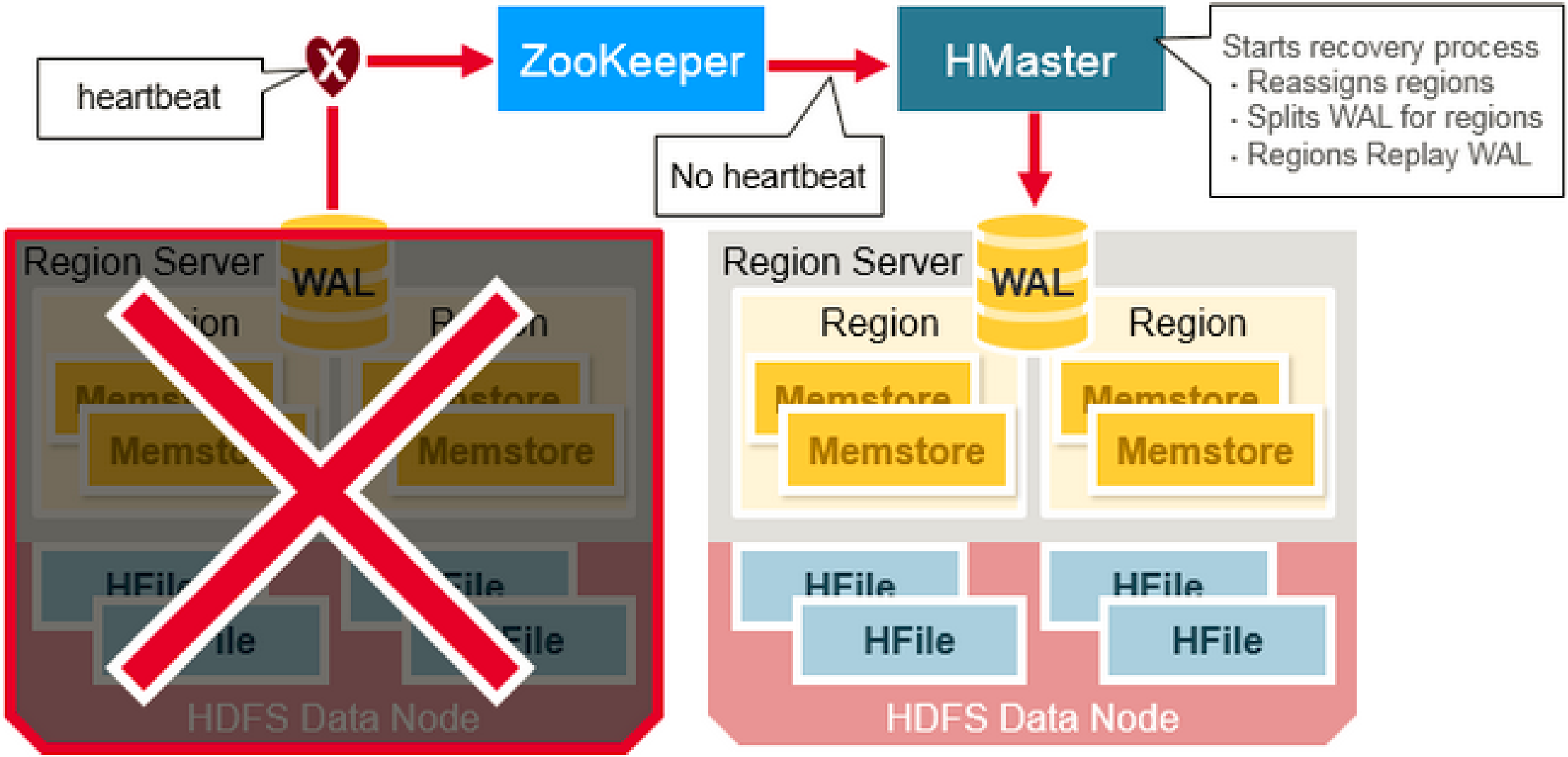


HDFS Data Replication (2)

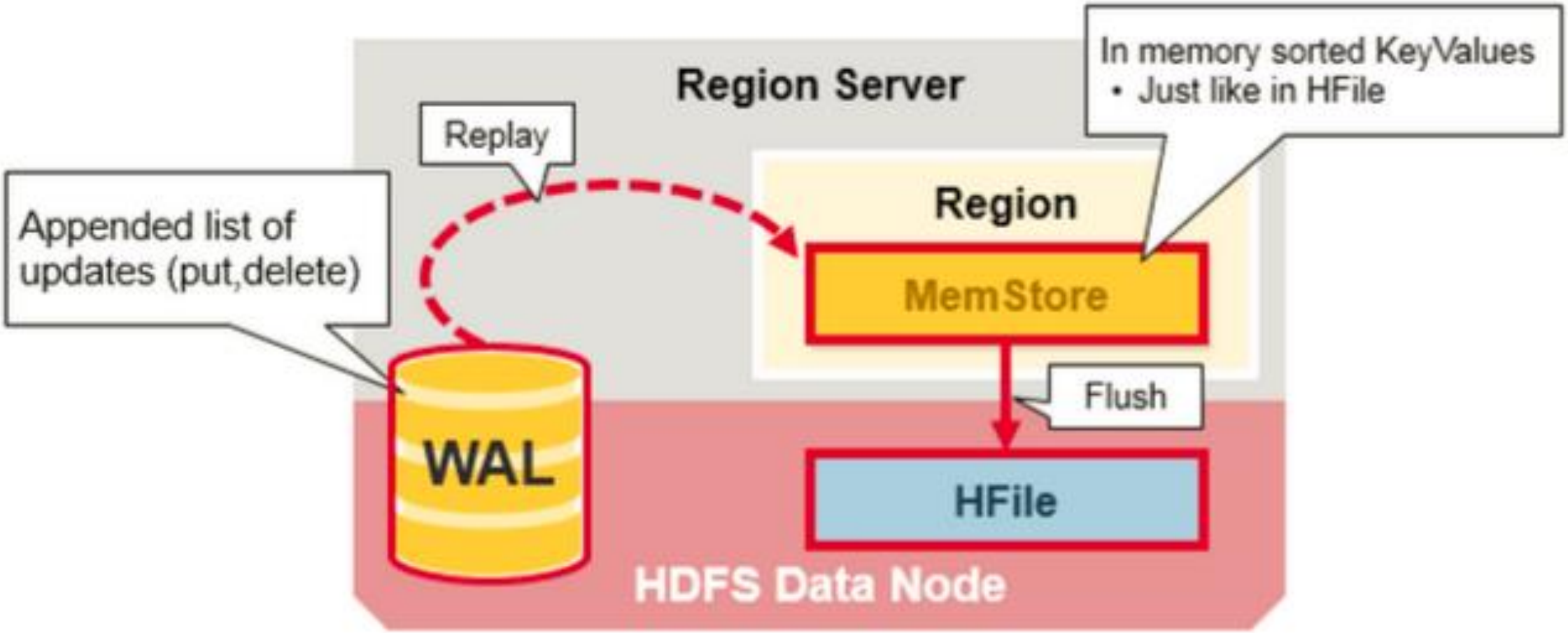


How does HBase recover updates not persisted to HFiles?

HBase Crash Recovery



Data Recovery



Apache HBase Architecture Benefits

HBase provides the following benefits:

- Strong consistency model- When a write returns, all readers will see same value
- Scales automatically- Regions split when data grows too large

- Uses HDFS to spread and replicate data

- Built-in recovery- Using Write Ahead Log (similar to journaling on file system)
- Integrated with Hadoop- MapReduce on HBase is straightforward

Apache HBase Has Problems Too...

Business continuity reliability:
- WAL replay slow
- Slow complex crash recovery
- Major Compaction I/O storms

目录

- 核心架构
- 参数调优
- 运维命令

参数调优：集群规划

硬盘容量敏感型业务：

这类业务对读写延迟以及吞吐量都没有很大的要求，唯一的需要就是硬盘容量。比如大多数离线读写分析业务，上层应用一般每隔一段时间批量写入大量数据，然后读取也是定期批量读取大量数据。特点：离线写、离线读，需求硬盘容量

带宽敏感型业务：

这类业务大多数写入吞吐量很大，但对读取吞吐量没有什么要求。比如日志实时存储业务，上层应用通过kafka将海量日志实时传输过来，要求能够实时写入，而读取场景一般是离线分析或者在上次业务遇到异常的时候对日志进行检索。特点：在线写、离线读，需求带宽- Slow complex crash recovery

IO敏感型业务：

相比前面两类业务来说，IO敏感型业务一般都是较为核心的业务。这类业务对读写延迟要求较高，尤其对于读取延迟通常在100ms以内，部分业务可能要求更高。比如在线消息存储系统、历史订单系统、实时推荐系统等。特点：在（离）线写、在线读，需求内存、高IOPS介质

HBase本身就是CPU敏感型系统，主要用于数据块的压缩/解压缩，因此对于HBase来说，CPU越多越好。所有业务都对CPU有共同的需求

建议不要将同一种类型的业务太多分布在同一个集群。一个集群理论上资源利用率比较高效的配置为：硬盘敏感型业务 ＋ 带宽敏感型业务 ＋ IO敏感型业务

集群容量规划

提出一个’Disk / Java Heap Ratio’的概念，意思是说一台RegionServer上1bytes的Java内存大小需要搭配多大的硬盘大小最合理

Disk Size / Java Heap = RegionSize / MemstoreSize * ReplicationFactor * HeapFractionForMemstore * 2

默认配置，RegionSize = 10G(hbase.hregion.max.filesize)
MemstoreSize = 128M（为hbase.hregion.memstore.flush.size）
ReplicationFactor = 3(dfs.replication)
HeapFractionForMemstore = 0.4（hbase.regionserver.global.memstore.lowerLimit）
计算为：10G / 128M * 3 * 0.4 * 2 = 192，意思是说RegionServer上1bytes的Java内存大小需要搭配192bytes的硬盘大小最合理，再回到之前给出的问题，128G的内存总大小，拿出96G作为Java内存用于RegionServer，那对应需要搭配96G ＊ 192 = 18T硬盘容量

推导过程：
硬盘容量纬度下Region个数：Disk Size / (RegionSize ＊ReplicationFactor)
Java Heap纬度下Region个数：Java Heap * HeapFractionForMemstore / (MemstoreSize / 2)

网络规划

带宽资源：因为HBase在大量scan以及高吞吐量写入的时候特别耗费网络带宽资源，强烈建议HBase集群部署在万兆交换机机房，单台机器最好也是万兆网卡＋bond。如果特殊情况交换机是千兆网卡，一定要保证所有的RegionServer机器部署在同一个交换机下，跨交换机会导致写入延迟很大，严重影响业务写入性能。

Region规划

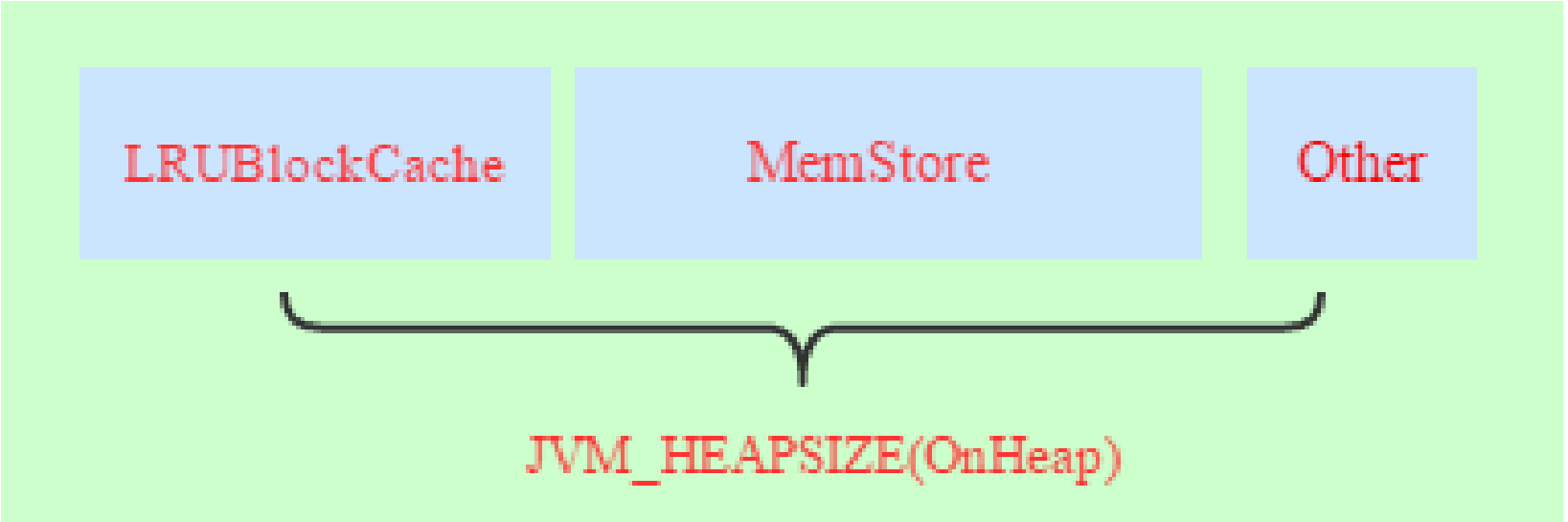
	优点	缺点
大量小Region	1. 更加有利于集群之间负载分布 2. 有利于高效平稳的Compaction，这是因为小Region中HFile相对较小，Compaction代价小，详情可见：Stripe Compaction	1. 最直接的影响：在某台RegionServer异常宕机或者重启的情况下大量小Region重分配以及迁移是一个很耗时的操作，一般一个Region迁移需要1.5s～2.5s左右，Region个数越多，迁移时间越长。直接导致failover时间很长。 2. 大量小Region有可能会产生更加频繁的flush，产生很多小文件，进而引起不必要的Compaction。特殊场景下，一旦Region数超过一个阈值，将会导致整个RegionServer级别的flush，严重阻塞用户读写。 3. RegionServer管理维护开销很大
少量大Region	1. 有利于RegionServer的快速重启以及宕机恢复 2. 可以减少总的RCP数量 3. 有利于产生更少的、更大的flush	1. Compaction效果很差，会引起较大的数据写入抖动，稳定性较差 2. 不利于集群之间负载均衡

官方文档给出的一个推荐范围一台RegionServer上的Region数在20～200之间，而单个Region大小控制在10G~30G，比较符合实际情况，Region数最直接取决于RegionSize的大小配置hbase.hregion.max.filesize，从Region规模这个角度讲，当前单台RegionServer能够合理利用起来的硬盘容量上限基本为18T：200 * 30G * 3＝ 18T

参数调优：内存规划

- 读缓存BlockCache
  - 读多写少型+BucketCache
  - 写多读少型+LRUBlockCache
- 写缓存MemStore

写多读少型 + LRUBlockCache



分配给RegionServer进程的内存就是JVM内存，主要分为三部分：LRUBlockCache，用于读缓存；MemStore，用于写缓存；Other，用于RS运行所必须的其他对象；

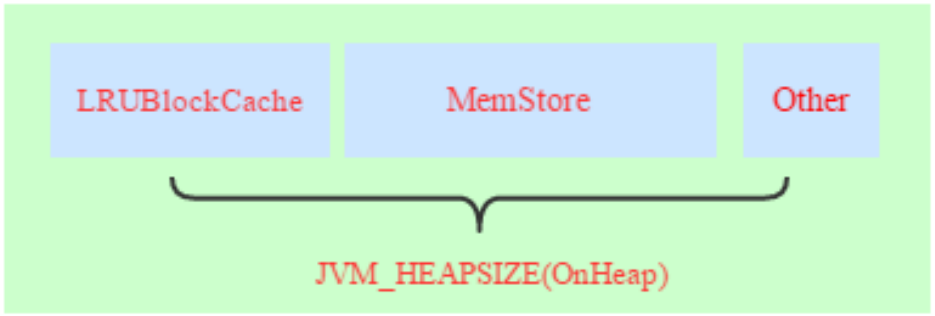
写多读少型 + LRUBlockCache

基本条件：
a. 整个物理机内存：96G
b. 业务负载分布：30%读，70%写

(1) 系统内存基础上如何规划RS内存？
这个问题需要根据自身服务器情况决定，一般情况下，在不影响其他服务的情况下，越大越好。我们目前设置为64G，为系统内存的2/3。

(2) 如何设置LRUBlockCache、MemStore？
确定RegionServer总内存之后，接下来分别规划LRUBlockCahce和MemStore的总内存。在此需要考虑两点：在写多读少的业务场景下，写缓存显然应该分配更多内存，读缓存相对分配更少；HBase在此处有个硬规定：LRUBlockCache + MemStore < 80% * JVM_HEAP，否则RS无法启动。
推荐内存规划：MemStore = 45% * JVM_HEAP = 64G * 45% = 28.8G ， LRUBlockCache = 30% * JVM_HEAP = 64G * 30% = 19.2G；默认情况下Memstore为40% * JVM_HEAP，而LRUBlockCache为25% * JVM_HEAP

写多读少型 + LRUBlockCache



（1）设置JVM参数如下：

```
-XX:SurvivorRatio=2 -XX:+PrintGCDateStamps
-Xloggc:$HBASE_LOG_DIR/gc-regionserver.log
-XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=1
-XX:GCLogFileSize=512M
 -server -Xmx64g -Xms64g -Xmn4g -Xss256k -XX:PermSize=256m -XX:MaxPermSize=256m
-XX:+UseParNewGC -XX:MaxTenuringThreshold=15
-XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection
-XX:+CMSClassUnloadingEnabled -XX:+UseCMSInitiatingOccupancyOnly
 -XX:CMSInitiatingOccupancyFraction=75 -XX:-DisableExplicitGC
```

写多读少型 + LRUBlockCache

（2）hbase-site.xml中MemStore相关参数设置如下

```
<property>
<name>hbase.regionserver.global.memstore.upperLimit</name>
<value>0.45</value>
</property>
<property>
<name>hbase.regionserver.global.memstore.lowerLimit</name>
<value>0.40</value>
</property>
```

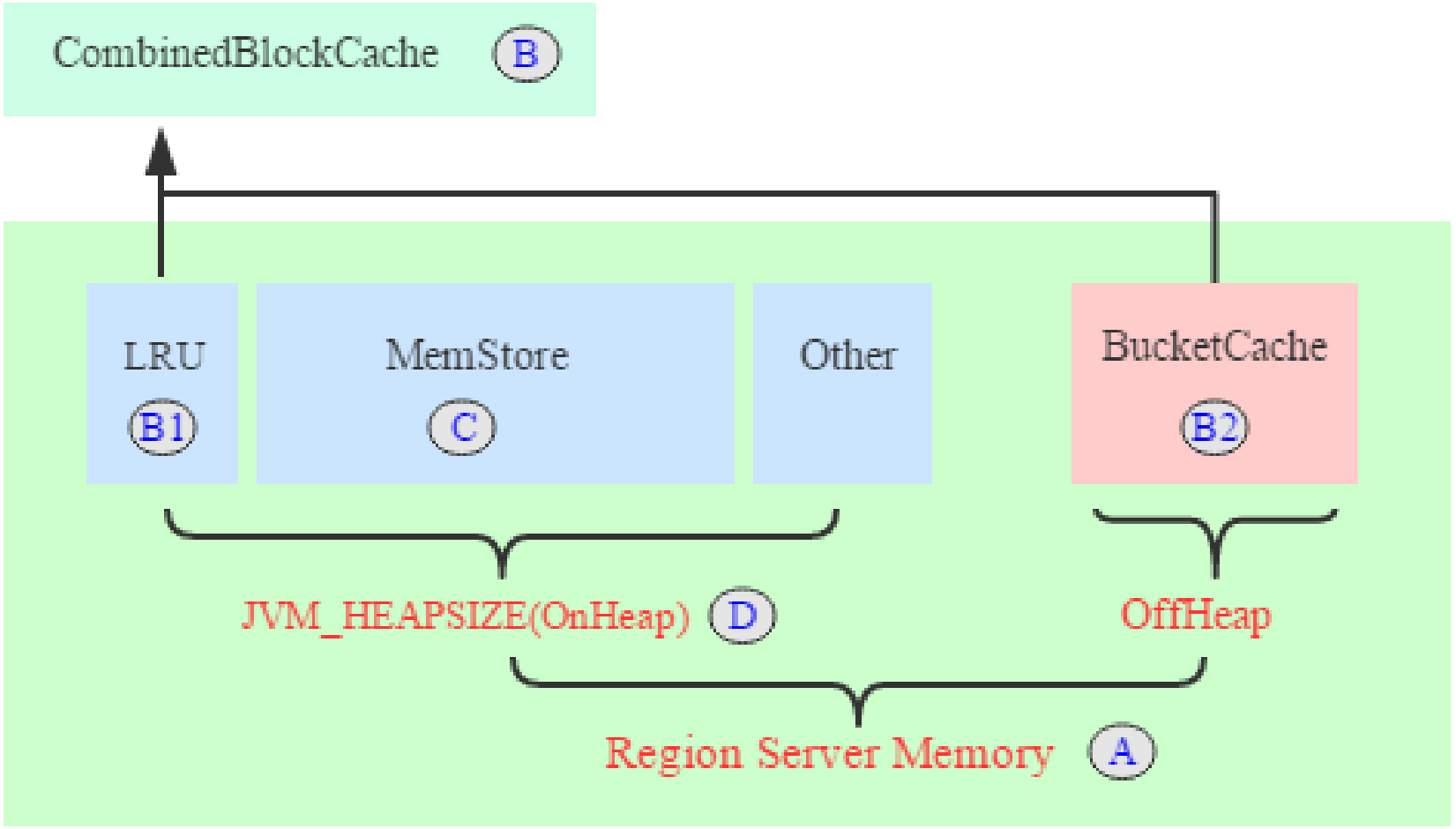
hbase.regionserver.global.memstore.upperLimit表示RegionServer中所有MemStore占有内存在JVM内存中的比例上限。如果所占比例超过这个值，RS会首先将所有Region按照MemStore大小排序，并按照由大到小的顺序依次执行flush，直至所有MemStore内存总大小小于hbase.regionserver.global.memstore.lowerLimit，一般lowerLimit比upperLimit小5%

（3）hbase-site.xml中LRUBlockCache相关参数设置如下：

```
<property>
<name>hfile.block.cache.size</name>
<value>0.3</value>
</property>
```

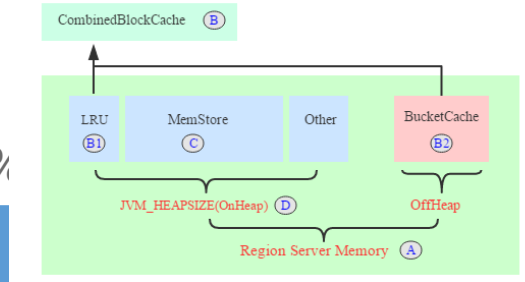
hfile.block.cache.size表示LRUBlockCache占用内存在JVM内存中的比例，因此设置为0.3

读多写少型 + BucketCache



整个RegionServer内存（Java进程内存）分为两部分：JVM内存和堆外内存。其中JVM内存中LRUBlockCache和堆外内存BucketCache一起构成了读缓存CombinedBlockCache，用于缓存读到的Block数据，其中LRUBlockCache用于缓存元数据Block，BucketCache用于缓存实际用户数据Block；MemStore用于写流程，缓存用户写入KeyValue数据；还有部分用于RegionServer正常运行所必须的内存；

读多写少型 + BucketCache



本案例中物理机内存也是96G，不过业务类型为读多写少：70%读+30%

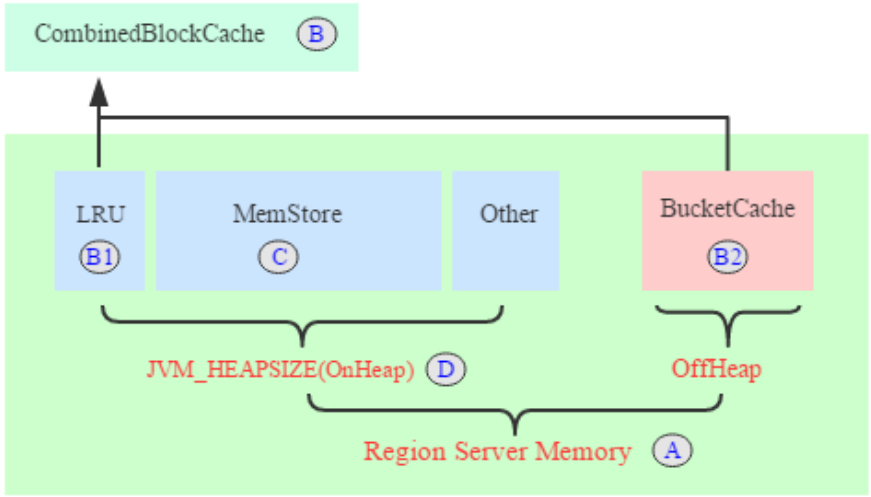
序号	步骤	原理	计算公式	计算值	修正值
A	规划RS总内存	在系统内存允许且不影响其他服务的情况下，越多越好。设置为系统总内存的 2/3。	2/3 * 96G	64G	64G
B	规划读缓存 CombinedBlockCache	整个RS内存分为三部分：读缓存、写缓存、其他。基本按照5 : 4 : 1的分配原则。读缓存设置为整个RS内存的50%	A * 50%	32G	26G
B1	规划读缓存LRU部分	LRU部分主要缓存数据块元数据，数据量相对较小。设置为整个读缓存的10%	B * 10%	3.2G	3G
B2	规划读缓存 BucketCache部分	BucketCache部分主要缓存用户数据块，数据量相对较大。设置为整个读缓存的90%	B * 90%	28.8G	24G
C	规划写缓存 MemStore	整个RS内存分为三部分：读缓存、写缓存、其他。基本按照5:4:1的分配原则。写缓存设置为整个RS内存的40%	A * 40%	25.6G	25G
D	设置JVM_HEAP	RS总内存大小 – 堆外内存大小	A – B2	35.2G	40G

LRU + MemStore / JVM_HEAP = 3G + 25G / 35G = 28G / 35G = 80%

建议这个值在70%~75%之间，因此需要对计算值进行简单的修正，适量增大JVM_HEAP值（增大至40G），BucketCache减少到24G（CombinedBlockCache需调整到26G）。调整之后，LRU + MemStore / JVM_HEAP = 3G + 25G / 40G = 28G / 40G = 70%

读多写少型 + BucketCache

（1）设置JVM参数如下：

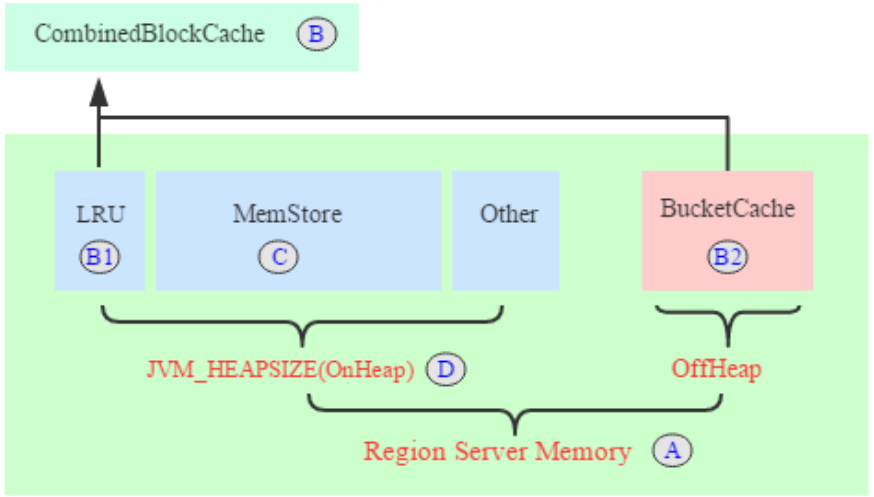



```
-XX:SurvivorRatio=2 -XX:+PrintGCDateStamps
-Xloggc:$HBASE_LOG_DIR/gc-regionserver.log
-XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=1 -XX:GCLogFileSize=512M
-server -Xmx40g -Xms40g –Xmn1g -Xss256k -XX:PermSize=256m -XX:MaxPermSize=256m
-XX:+UseParNewGC -XX:MaxTenuringThreshold=15 -XX:+CMSParallelRemarkEnabled
-XX:+UseCMSCompactAtFullCollection -XX:+CMSClassUnloadingEnabled
-XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=75
-XX:-DisableExplicitGC
```

读多写少型 + BucketCache

（2）hbase-site.xml中MemStore相关参数设置如下：

```
<property>
<name>hbase.regionserver.global.memstore.upperLimit</name>
<value>0.60</value>
</property>
<property>
<name>hbase.regionserver.global.memstore.lowerLimit</name>
<value>0.55</value>
</property>
```



upperLimit = 25G / 40G = 60%。因此upperLimit参数设置为0.60，lowerLimit设置为0.55

（3）hbase-site.xml中MemStore相关参数设置如下：

```
<property>
<name>hbase.bucketcache.ioengine</name>
<value>offheap</value>
</property>
<property>
<name>hbase.bucketcache.size</name>
<value>26624</value>
</property>
<property>
<name>hbase.bucketcache.percentage.in.combinedcache</name>
<value>0.90</value>
</property>
```

hbase.bucketcache.ioengine表示bucketcache设置为offheap模式；hbase.bucketcache.size表示所有读缓存占用内存大小，该值可以为内存真实值，单位为M，也可以为比例值，表示读缓存大小占JVM内存大小比例。如果为内存真实值，则为26G，即26624M。而如果是比例值，则计算应为 26G / 40G = 65%，设为0.65即可；hbase.bucketcache.percentage.in.combinedcache参数表示用于缓存用户数据块的内存（堆外内存）占所有读缓存的比例，设为0.90；

参数调优：客户端重试机制

1. hbase.client.pause:默认为100，可以减少为50
2. hbase.client.retries.number:默认为31，可以减少为21

参数调优：客户端超时机制

hbase.rpc.timeout

表示一次RPC请求的超时时间。如果某次RPC时间超过该值，客户端就会主动关闭socket。此时，服务器端就会捕获到如下的异常：java.io.IOException: Connection reset by peer，异常经常发生在大量高并发读写业务或者服务器端发生了比较严重的Full GC等场景下，导致某些请求无法得到及时处理，超过了时间间隔。该值默认大小为60000ms，即1min

hbase.client.operation.timeout

该参数表示HBase客户端发起一次数据操作直至得到响应之间总的超时时间，数据操作类型包括get、append、increment、delete、put等。很显然，hbase.rpc.timeout表示一次RPC的超时时间，而hbase.client.operation.timeout则表示一次操作的超时时间，有可能包含多个RPC请求。举个例子说明，比如一次Put请求，客户端首先会将请求封装为一个caller对象，该对象发送RPC请求到服务器，假如此时因为服务器端正好发生了严重的Full GC，导致这次RPC时间超时引起SocketTimeoutException，对应的就是hbase.rpc.timeout。那假如caller对象发送RPC请求之后刚好发生网络抖动，进而抛出网络异常，HBase客户端就会进行重试，重试多次之后如果总操作时间超时引起SocketTimeoutException，对应的就是hbase.client.operation.timeout。

hbase.client.scanner.timeout.period

hbase.client.operation.timeout参数规定的超时基本涉及到了HBase所有的数据操作，唯独没有scan操作。然而scan操作却是最有可能发生超时的，也因此是用户最为关心的。HBase当然考虑到了这点，并提供了一个单独的超时参数进行设置：hbase.client.scanner.timeout.period

常用参数：通用和master配置[1]

参数	默认值	说明
hbase.rootdir	file:///tmp/hbase-${user.name}/hbase	region server的数据根目录，用来持久化HBase。例如，要表示hdfs中的’/hbase’目录，namenode 运行在debugo01的8020端口,则需要设置为hdfs:// debugo01:8020/hbase。这个是必须要设置的项目，默认值本地文件系统的/tmp只能在单机模式使用。
hbase.master.port	60000	HBase的Master服务端口
hbase.cluster.distributed	false	HBase的运行模式。false是单机模式，true是分布式模式。若为false,HBase和Zookeeper会运行在同一个JVM里面。
hbase.tmp.dir	/tmp/hbase-${user.name}	本地的临时目录。需要注意的是默认的/tmp会在重启时清空
hbase.master.info.port	60010	HBase Master web UI的端口. 设置为-1 则不运行web UI
hbase.master.info.bindAddress	0.0.0.0	HBase Master web UI绑定的IP
hbase.master.dns.interface	default	当使用DNS的时候，Master用来上报的IP地址的网络接口名字
hbase.master.dns.nameserver	default	当使用DNS的时候，RegionServer的DNS域名或者IP 地址，Master用它来确定用来进行通讯的域名.

常用参数：通用和master配置[2]

参数	默认值	说明
hbase.snapshot.enabled	true	是否启用snapshot功能
hbase.balancer.period	300000	Master执行region balancer的间隔
hbase.master.logcleaner.ttl	600000	Hlog存在于.oldlogdir 文件夹的最长时间, 超过了就会被 Master 的线程清理掉
hbase.master.logcleaner.plugins	org.apache.hadoop.hbase.master.TimeToLiveLogCleaner	LogsCleaner服务会执行的一组LogCleanerDelegat。值用逗号间隔的文本表示。这些WAL/HLog cleaners会按顺序调用。可以把先调用的放在前面。你可以实现自己的LogCleanerDelegat，加到Classpath下，然后在这里写下类的全称。一般都是加在默认值的前面。
hbase.rest.port	8080	HBase REST server的端口
hbase.rest.readonly	false	定义REST server的运行模式。可以设置成如下的值： false: 所有的HTTP请求都是被允许的 – GET/PUT/POST/DELETE. true:只有GET请求是被允许的
hbase.coprocessor.master.classes	null	Master所使用的协处理器类名。多个类的情况下使用逗号分割。比如使用org.apache.hadoop.hbase.security.access.AccessController提供了安全管控能力。
hbase.rpc.engine	null	HBase使用的rpc引擎类。一般使用带安全验证的org.apache.hadoop.hbase.ipc. SecureRpcEngine类

常用参数：RegionServer相关[1]

参数	默认值	说明
hbase.regionserver.port	60020	HBase RegionServer绑定的端口
hbase.regionserver.info.port	60030	HBase RegionServer web 界面绑定的端口。设置为-1 则不运行web UI
hbase.regionserver.info.port.auto	false	RegionServer是否在当hbase.regionsever.info.port已经被占用的时候，可以搜一个空闲的端口绑定。默认关闭。
hbase.regionserver.info.bindAddress	0.0.0.0	HBase RegionServer web 界面的IP地址
hbase.regionserver.class	org.apache.hadoop.hbase.ipc.HRegionInterface	RegionServer接口类。客户端在连接region server的时候会使用到
hbase.regionserver.lease.period	60000	客户端租用HRegion server 期限，即超时阀值。单位是毫秒。默认情况下，客户端必须在这个时间内发一条信息，否则视为死掉。
hbase.regionserver.handler.count	10	RegionServers受理的RPC Server实例数量。对于Master来说，这个属性是Master受理的handler数量
hbase.regionserver.msginterval	3000	RegionServer 发消息给 Master 时间间隔，单位是毫秒
hbase.regionserver.optionallogflushinterval	1000	将Hlog同步到HDFS的间隔。如果Hlog没有积累到一定的数量，到了时间，也会触发同步。默认是1秒，单位毫秒
hbase.regionserver.regionSplitLimit	2147483647	region的数量到了这个值后就不会在分裂了。这不是一个region数量的硬性限制。但是起到了一定指导性的作用。默认是MAX_INT（不限制）
hbase.regionserver.hlog.enabled	true	是否启用WAL

常用参数：RegionServer相关[2]

参数	默认值	说明
hbase.regionserver.wal.enablecompression	false	是否对日志压缩
hbase.regionserver.hlog.reader.impl		HLog file reader 的实现,默认 org.apache.hadoop.hbase.regionserver.wal.SequenceFileLogReader
hbase.regionserver.hlog.writer.impl		HLog file writer 的实现，默认 org.apache.hadoop.hbase.regionserver.wal.SequenceFileLogWriter
hbase.regionserver.thread.splitcompactcheckfrequency	20000	region server 多久执行一次split/compaction 检查
hbase.regionserver.nbreservationblocks	4	储备的内存block的数量(译者注:就像石油储备一样)。当发生out of memory 异常的时候，我们可以用这些内存在RegionServer停止之前做清理操作
hbase.regionserver.dns.interface	default	当使用DNS的时候，RegionServer用来上报的IP地址的网络接口名字
hbase.regionserver.dns.nameserver	default	当使用DNS的时候，RegionServer使用的DNS的域名或者IP 地址，RegionServer用它来确定和master用来进行通讯的域名.
hbase.regions.slop	0	当任一regionserver有average + (average * slop)个region是会执行Rebalance

常用参数：RegionServer相关[3]

参数	默认值	说明
hbase.regionserver.global.memstore.upperLimit	0.4	单个region server的全部memtores的最大值。超过这个值，一个新的update操作会被挂起，强制执行flush操作
hbase.regionserver.global.memstore.lowerLimit	0.35	当强制执行flush操作的时候，当低于这个值的时候，flush会停止。默认是堆大小的 35% . 如果这个值和hbase.regionserver.global.memstore.upperLimit 相同就意味着当update操作因为内存限制被挂起时，会尽量少的执行flush(译者注:一旦执行flush，值就会比下限要低，不再执行)
hbase.coprocessor.regionserver.classes	null	regionserver所使用的协处理器类名。多个类的情况下使用逗号分割。比如使用org.apache.hadoop.hbase.security.access.AccessController提供了安全管控能力

常用参数：Client相关参数

参数	默认值	说明
hbase.client.write.buffer	2097152	HTable客户端的写缓冲的默认大小。因为缓冲在客户端和服务端需要申请空间，需要消耗客户端和服务端两个地方的内存。较大的buffer，可以减少RPC的次数，从而提高性能。估算服务器端被占用的内存：hbase.client.write.buffer * hbase.regionserver.handler.count
hbase.client.pause	1000	客户端操作失败后的重试暂停时间
hbase.client.retries.number	10	客户端失败后的最大重试次数。例如region查询，get，Update等操作
hbase.client.scanner.caching	1	当调用Scanner的next方法，而值又不在缓存里的时候，从服务端一次获取的行数。越大的值意味着Scanner会快一些，但是会占用更多的内存。当缓冲被占满的时候，next方法调用会越来越慢。慢到一定程度，可能会导致超时。例如超过了hbase.regionserver.lease.period。
hbase.client.keyvalue.maxsize	10485760	一个KeyValue实例的最大size.这个是用来设置存储文件中的单个entry的大小上界。因为一个KeyValue是不能分割的，所以可以避免因为数据过大导致region不可分割。明智的做法是把它设为可以被最大region size整除的数。如果设置为0或者更小，就会禁用这个检查。默认10MB。

常用参数：ZooKeeper相关[1]

参数	默认值	说明
hbase.zookeeper.dns.interface	default	当使用DNS的时候，Zookeeper用来上报的IP地址的网络接口名字
hbase.zookeeper.dns.nameserver	default	当使用DNS的时候，Zookeepr使用的DNS的域名或者IP 地址，Zookeeper用它来确定和master用来进行通讯的域名
zookeeper.session.timeout	180000	ZooKeeper 会话超时.HBase把这个值传递改zk集群，向他推荐一个会话的最大超时时间。单位是毫秒
zookeeper.znode.parent	/hbase	ZooKeeper中的HBase的根ZNode。所有的HBase的ZooKeeper会用这个目录配置相对路径。默认情况下，所有的HBase的ZooKeeper文件路径是用相对路径，所以他们会都去这个目录下面。
zookeeper.znode.rootserver	root-region-server	ZNode 保存的根region的路径. 这个值是由Master来写，client和regionserver 来读的。如果设为一个相对地址，父目录就是${zookeeper.znode.parent}.默认情形下，意味着根region的路径存储在/hbase/root-region-server.
hbase.zookeeper.quorum	localhost	Zookeeper集群的地址列表，用逗号分割。例如：”host1.mydomain.com,host2.mydomain.com,host3.mydomain.com”.默认是localhost,如果在hbase-env.sh设置了HBASE_MANAGES_ZK，这些ZooKeeper节点就会和HBase一起启动。
hbase.zookeeper.peerport	2888	ZooKeeper节点使用的端口。

常用参数：ZooKeeper相关[2]

参数	默认值	说明
hbase.zookeeper.leaderport	3888	ZooKeeper用来选择Leader的端口
hbase.zookeeper.property.initLimit	10	ZooKeeper的zoo.conf中的配置。 初始化synchronization阶段的ticks数量限制
hbase.zookeeper.property.syncLimit	5	ZooKeeper的zoo.conf中的配置。 发送一个请求到获得承认之间的ticks的数量限制
hbase.zookeeper.property.dataDir	${hbase.tmp.dir}/zookeeper	ZooKeeper的zoo.conf中的配置。 快照的存储位置
hbase.zookeeper.property.clientPort	2181	ZooKeeper的zoo.conf中的配置。 客户端连接的端口
hbase.zookeeper.property.maxClientCnxns	2000	ZooKeeper的zoo.conf中的配置。 ZooKeeper集群中的单个节点接受的单个Client(以IP区分)的请求的并发数。这个值可以调高一点，防止在单机和伪分布式模式中出问题。

常用参数：schema和存储相关[1]

参数	默认值	说明
hbase.table.max.rowsize	true	设置为true时，当schema在反生变更操作时则锁定zookeeper中的表，以避免并行操作时破坏表的一致性。
hbase.table.max.rowsize	1073741824	当不指定Get和Scan操作中的scan选项时，获取的最大行的字节大小。超过则抛出RowTooBigException异常。
hbase.hregion.memstore.flush.size	67108864	当memstore的大小超过这个值，会flush到磁盘。这个值被一个线程每隔hbase.server.thread.wakefrequency检查一下。
hbase.server.thread.wakefrequency	10000	Service线程（log roller等）的sleep时间间隔，单位毫秒
hbase.hregion.preclose.flush.size	5242880	当一个region中的memstore的大小大于这个值的时候，我们又触发了close.会先运行“pre-flush”操作，清理这个需要关闭的memstore，然后将这个region下线。当一个region下线了，我们无法再进行任何写操作。如果一个memstore很大的时候，flush操作会消耗很多时间。”pre-flush”操作意味着在region下线之前，会先把memstore清空。这样在最终执行close操作的时候，flush操作会很快。
hbase.hregion.memstore.block.multiplier	2	如果memstore有hbase.hregion.memstore.block.multiplier倍数的hbase.hregion.flush.size的大小，就会阻塞update操作。这是为了预防在update高峰期会导致的失控。如果不设上界，flush的时候会花很长的时间来合并或者分割，最坏的情况就是引发out of memory异常。
hbase.hregion.memstore.mslab.enabled	false	体验特性：启用memStore分配本地缓冲区。这个特性是为了防止在大量写负载的时候堆的碎片过多。这可以减少GC操作的频率。

常用参数： schema和存储相关[2]

参数	默认值	说明
hbase.hregion.max.filesize	268435456	最大HStoreFile大小。若某个Column families的HStoreFile增长达到这个值，这个Hegion会被切割成两个。 Default: 256M
hbase.hstore.compactionThreshold	3	当一个HStore含有多于这个值的HStoreFiles的时候，会执行一个合并操作，把这HStoreFiles写成一个。这个值越大，需要合并的时间就越长。
hbase.hstore.blockingStoreFiles	7	当一个HStore含有多于这个值的HStoreFiles的时候，会执行一个合并操作，update会阻塞直到合并完成，直到超过了hbase.hstore.blockingWaitTime的值
hbase.hstore.blockingWaitTime	90000	hbase.hstore.blockingStoreFiles所限制的StoreFile数量会导致update阻塞，这个时间是来限制阻塞时间的。当超过了这个时间，HRegion会停止阻塞update操作，不过合并还有没有完成。默认为90s.
hbase.hstore.compaction.max	10	每个“小”合并的HStoreFiles最大数量。
hbase.hregion.majorcompaction	86400000	一个Region中的所有HStoreFile的major compactions的时间间隔。默认是1天。 设置为0就是禁用这个功能
hbase.mapreduce.hfileoutputformat.blocksize	65536	MapReduce中HFileOutputFormat可以写 storefiles/hfiles. 这个值是hfile的blocksize的最小值。通常在HBase写Hfile的时候，bloocksize是由table schema(HColumnDescriptor)决定的，但是在mapreduce写的时候，我们无法获取schema中blocksize。这个值越小，你的索引就越大，你随机访问需要获取的数据就越小。如果你的cell都很小，而且你需要更快的随机访问，可以把这个值调低。
hfile.block.cache.size	0.2	分配给HFile/StoreFile的block cache占最大堆(-Xmx setting)的比例。默认是20%，设置为0就是不分配。
hbase.hash.type	murmur	哈希函数使用的哈希算法。可以使用MurmurHash和 jenkins (JenkinsHash). 提供给 bloom filters使用.

常用参数：安全相关

参数	默认值	说明
hbase.security.authentication	simple	HBase的权限管控方式。可选输入为simple和kerberos。
hbase.security.authentication	false	是否启用HBase安全验证
hbase.superuser	hbase	HBase的超级管理员用户
hbase.master.keytab.file		开启基于kerberos的Security。HMaster Server验证登录使用的keytab 文件路径
hbase.master.kerberos.principal		开启基于kerberos的Security。HMaster使用principal name
hbase.regionserver.keytab.file		开启基于kerberos的Security。HRegionServer验证登录使用的keytab 文件路径
hbase.regionserver.kerberos.principal		开启基于kerberos的Security。RegionServer使用的principal name。

目录

- 核心架构
- 参数调优
- 运维命令

集群的启动与停止

hadoop zookeeper必须在hbase 之前启动，但是hadoop和zookeeper并没有先后顺序要求。

- 启动：
  Hadoop启动：$HADOOP_HOME\sbin\start-all.sh
  Zookeeper启动：$ZOOKEEPER_HOME\bin\zkServer.sh start
  Hbase启动：$HBASE_HOME\sbin\start-hbase.sh

- 停止：
  Hbase停止：$HBASE_HOME\sbin\stop-hbase.sh
  Zookeeper停止：$ZOOKEEPER_HOME\bin\zkServer.sh stop
  Hadoop停止：$HADOOP_HOME\sbin\stop-all.sh

Hbase集群数据清空操作

HBase Shell 常用操作：分类

分类	命令
一般性命令	status, version
DDL[数据表定义]	alter, create, describe, disable,drop, enable, exists, is_disabled, is_enabled, list
DML[数据操作]	count, delete, deleteall, get,get_counter, incr, put, scan, truncate
工具	assign, balance_switch, balancer,close_region, compact, flush, major_compact, move, split, unassign, zk_dump
灾备[复制replication]	add_peer, disable_peer,enable_peer, remove_peer, start_replication, stop_replication
灾备[快照snapshot]	clone_snapshot, delete_snapshot, list_snapshots, restore_snapshot, snapshot
security	grant, revoke, user_permission

HBase Shell 常用操作：一般性命令

```
[hadoop@hbase bin]$ ./hbase shell
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/hadoop/PAI/hbase/lib/slf4j-log4j12-1.7.5.jar!/org/s
inder.class]
SLF4J: Found binding in [jar:file:/home/hadoop/PAI/hadoop/share/hadoop/common/lib/slf4j-log
4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.1.3, rUnknown, Wed May 18 16:25:19 CST 2016

hbase(main):001:0> status
1 servers, 0 dead, 85.0000 average load

hbase(main):002:0> version
1.1.3, rUnknown, Wed May 18 16:25:19 CST 2016
```

HBase Shell 常用操作：DDL--Create

- **建表 Create：** create table_name, column_family1, column_family2,…
  - 这个时候这个表是创建在default下面。如果需要在debugo_ns这个命名空间下面建表，则需要使用create namespace:table_name这种方式

```
> create 'user','info'
0 row(s) in 0.9030 seconds
=> Hbase::Table - user
```

```
> create_namespace 'debugo_ns'
0 row(s) in 2.0910 seconds
create 'debugo_ns:users', 'info'
0 row(s) in 0.4640 seconds
=> Hbase::Table - debugo_ns:users
```

每个列簇中可以独立指定它使用的版本数，数据有效保存时间（TTL），是否开启块缓存等信息。

```
> create 't1', {NAME => 'f1', VERSIONS => 1, TTL => 2592000, BLOCKCACHE => true}, 'f2'
```

表也可以在创建时指定它预分割(pre-splitting)的region数和split方法。在表初始建立时，HBase只分配给这个表一个region。这就意味着当我们访问这个表数据时，我们只会访问一个region server，这样就不能充分利用集群资源。

```
> create 't2', 'f1', {NUMREGIONS => 3, SPLITALGO => 'HexStringSplit'}
```

HBase Shell 常用操作： DDL– describe, enable, disable, is_enabled, is_disabled

- 通过describe 来查看这个表中的元信息

```
> describe 't2'
DESCRIPTION                                                    ENABLED
 't2', {NAME => 'f1', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLIC tru
1 row(s) in 0.0690 seconds
```

- 通过enable和disable来启用/禁用这个表,相应的可以通过is_enabled和is_disabled来检查表是否被禁用

```
> disable 't2'
0 row(s) in 1.4410 seconds
> enable 't2'
0 row(s) in 0.5940 seconds
> is_enabled 't2'
true
0 row(s) in 0.0400 seconds
hbase(main):042:0> is_disabled 't2'
false
0 row(s) in 0.0490 seconds
```

- 使用exists来检查表是否存在

```
> exists 't2'
Table t2 does exist
0 row(s) in 0.0590 seconds
```

HBase Shell 常用操作： DDL– alter,drop,list

- 使用alter来改变表的属性，比如改变列簇的属性, 这涉及将信息更新到所有的region,目前版本不需要先把table禁用

```
> alter 't1', {NAME => 'f1', VERSIONS => 5}
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
0 row(s) in 2.3470 seconds
```

- 使用alter来添加和删除列簇

```
> alter 't1','f3'
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
0 row(s) in 2.3130 seconds
> alter 't1', 'delete' => 'f3'
```

```
> alter 't1',{ NAME => 'f3', METHOD => 'delete'}
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
0 row(s) in 2.2930 seconds
```

- 使用drop删除表

```
> disable 't1'
0 row(s) in 1.4310 seconds
> drop 't1'
0 row(s) in 0.2440 seconds
```

- 使用list罗列所有表

HBase Shell 常用操作： DML– put,get

- 通过put命令来插入数据,列簇下的列不需要提前创建，在需要时通过:来指定即可

```
> create 'member','id','address','info'
0 row(s) in 0.4570 seconds
=> Hbase::Table - member
put 'member', 'debugo','id','11'
put 'member', 'debugo','info:age','27'
put 'member', 'debugo','info:birthday','1987-04-04'
put 'member', 'debugo','info:industry', 'it'
put 'member', 'debugo','address:city','beijing'
put 'member', 'debugo','address:country','china'
put 'member', 'Sariel', 'id', '21'
put 'member', 'Sariel','info:age', '26'
put 'member', 'Sariel','info:birthday', '1988-05-09 '
put 'member', 'Sariel','info:industry', 'it'
put 'member', 'Sariel','address:city', 'beijing'
put 'member', 'Sariel','address:country', 'china'
put 'member', 'Elvis', 'id', '22'
put 'member', 'Elvis','info:age', '26'
put 'member', 'Elvis','info:birthday', '1988-09-14 '
put 'member', 'Elvis','info:industry', 'it'
put 'member', 'Elvis','address:city', 'beijing'
put 'member', 'Elvis','address:country', 'china'
```

- 通过get命令来获得数据，如获取一个id的所有数据

```
> get 'member', 'Sariel'
COLUMN                          CELL
 address:city                   timestamp=1425871035382, value=beijing
 address:country                timestamp=1425871035424, value=china
 id:                            timestamp=1425871035176, value=21
 info:age                       timestamp=1425871035225, value=26
 info:birthday                  timestamp=1425871035296, value=1988-05-09
 info:industry                  timestamp=1425871035334, value=it
6 row(s) in 0.0530 seconds
```

默认情况下列簇只保存1个version。我们先将其修改到2,然后update一些信息

```
> alter 'member', {NAME=> 'info', VERSIONS => 2}
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
0 row(s) in 2.2580 seconds
> put 'member', 'debugo','info:age','29'
> put 'member', 'debugo','info:age','28'
> get 'member', 'debugo', {COLUMN=>'info:age', VERSIONS=>2}
COLUMN                          CELL
 info:age                       timestamp=1425884510241, value=28
 info:age                       timestamp=1425884510195, value=29
2 row(s) in 0.0400 seconds
```

HBase Shell 常用操作：DML– delete，deleteall，incr，count

- 通过delete命令，我们可以删除id为某个值的‘info:age’字段

```
> delete 'member','debugo','info:age'
0 row(s) in 0.0420 seconds
> get 'member','debugo','info:age'
COLUMN                                CELL
0 row(s) in 0.3270 seconds
```

- 通过deleteall来删除整行

```
> delete 'member','debugo','info:age'
0 row(s) in 0.0420 seconds
> get 'member','debugo','info:age'
COLUMN                                CELL
0 row(s) in 0.3270 seconds
```

- 使用incr实现递增，这个value需要是一个数值

```
> delete 'member','Sariel','info:age'
0 row(s) in 0.0270 seconds
> incr 'member','Sariel','info:age',26
0 row(s) in 0.0290 seconds
> incr 'member','Sariel','info:age'
0 row(s) in 0.0290 seconds
> incr 'member','Sariel','info:age', -1
0 row(s) in 0.0230 seconds
> get 'member','Sariel','info:age'
COLUMN   CELL
 info:age   timestamp=1425890213341, value=\x00\x00\x00\x00\x00\x00\x00\x1A
1 row(s) in 0.0280 seconds
```

- 通过count统计行数

```
> count 'member'
2 row(s) in 0.0750 seconds
=> 2
```

HBase Shell 常用操作：DML– truncate， scan

- 通过truncate来截断表。hbase是先将掉disable掉，然后drop掉后重建表来实现truncate的功能的

```
hbase(main):010:0> truncate 'member'
Truncating 'member' table (it may take a while):
 - Disabling table...
 - Dropping table...
 - Creating table...
0 row(s) in 2.3260 seconds
```

- 通过scan来对全表进行扫描

```
> scan 'member'
ROW                   COLUMN+CELL
 Elvis                column=address:city, timestamp=1425891057211, value=
                      beijing
 Elvis                column=address:country, timestamp=1425891057258, val
                      ue=china
```

- 通过scan来扫描其中的某个列或者整个列簇

```
> scan 'member', {COLUMNS=> 'info:birthday'}
```

```
> scan 'member', {COLUMNS=> 'info'}
ROW                              COLUMN+CELL
 Elvis                           column=info:age, timestamp=1425891057083, value=26
```

HBase Shell 常用操作： DML– Limit, STARTROW , STOPROW, TIMERANGE, VERSIONS

- Limit（限制查询结果行数）
- STARTROW （ROWKEY起始行。会先根据这个key定位到region，再向后扫描）
- STOPROW(结束行)
- TIMERANGE（限定时间戳范围）
- VERSIONS（版本数）

比如我们从Sariel这个rowkey开始，找下一个行的最新版本：

```
> scan 'member', { STARTROW => 'Sariel', LIMIT=>1, VERSIONS=>1}
ROW    COLUMN+CELL
 Sariel    column=address:city, timestamp=1425891056965, value=beijing
 Sariel    column=address:country, timestamp=1425891057003, value=china
 Sariel    column=id:, timestamp=1425891056767, value=21
 Sariel    column=info:age, timestamp=1425891056808, value=26
 Sariel    column=info:birthday, timestamp=1425891056883, value=1988-05-09
 Sariel    column=info:industry, timestamp=1425891056924, value=it
1 row(s) in 0.0410 seconds
```

HBase Shell 常用操作：DML– 内置filter及扩展filter

- Filter是一个非常强大的修饰词，可以设定一系列条件来进行过滤

比如我们要限制某个列的值等于26

```
> scan 'member', FILTER=>"ValueFilter(=,'binary:26')"
ROW    COLUMN+CELL
 Elvis    column=info:age, timestamp=1425891057083, value=26
 Sariel   column=info:age, timestamp=1425891056808, value=26
2 row(s) in 0.0620 seconds
```

值包含6这个值

```
> scan 'member', FILTER=>"ValueFilter(=,'substring:6')"
Elvis    column=info:age, timestamp=1425891057083, value=26
 Sariel   column=info:age, timestamp=1425891056808, value=26
2 row(s) in 0.0620 seconds
```

列名中的前缀为birthday的

```
> scan 'member', FILTER=>"ColumnPrefixFilter('birth') "
ROW                              COLUMN+CELL
 Elvis                           column=info:birthday, timestamp=1425891057129, value
 Sariel      column=info:birthday, timestamp=1425891056883, value=1988-05-09
 debugo                          column=info:birthday, timestamp=1425891056547, value
3 row(s) in 0.0450 seconds
```

HBase Shell 常用操作：DML– 内置filter

分类	Filter名	作用
Comparision Filters	RowFilter	选择比较rowkey来确定选择合适的行信息，筛选出匹配的所有的行
FamilyFilter	FamilyFilter	通过和列簇比较得到，返回结果为真的数据
	QualifierFilter	通过和列限定符比较，返回为真的数据
	ValueFilter	按照具体的值来筛选单元格的过滤器，这会把一行中值不能满足的单元格过滤掉
	DependentColumnFilter	该过滤器有两个参数 —— 列族和列修饰。尝试找到该列所在的每一行，并返回该行具有相同时间戳的全部键值对。如果某一行不包含指定的列，则该行的任何键值对都不返回。 该过滤器还可以有一个可选布尔参数 —— dropDependentColumn. 如果为true,从属的列不返回。 该过滤器还可以有两个可选参数 —— 一个比较操作符和一个值比较器，用于列族和修饰的进一步检查。如果从属的列找到，其值还必须通过值检查，然后就是时间戳必须考虑
Dedicated Filters	SingleColumnValueFilter	选定列簇和某一列，然后与列的value相比，正确的返回全部的row，注意如果某一行不含有该列，同样返回，除非通过filterIfColumnMissing 设置成真
	SingleColumnValueExcludeFilter	该过滤器同上面的过滤器正好相反，如果条件相符，将不会返回该列的内容
	PrefixFilter	筛选出具有特定前缀的行键的数据。这个过滤器所实现的功能其实也可以由RowFilter结合RegexComparator来实现
	PageFilter	页过滤，通过设置pagesize参数可以返回每一页page的数量。客户端需要记住上一次访问的row的key值
	KeyOnlyFilter	这个过滤器唯一的功能就是只返回每行的行键，值全部为空，这对于只关注于行键的应用场景来说非常合适，这样忽略掉其值就可以减少传递到客户端的数据量，能起到一定的优化作用
	FirstKeyOnlyFilter	返回的结果集中只包含第一列的数据，它在找到每行的第一列之后会停止扫描，从而使扫描的性能也得到了一定的提升

HBase Shell 常用操作：DML– 内置filter

	Filter名	作用
Dedicated Filters	InclusiveStopFilter	扫描的时候，我们可以设置一个开始行键和一个终止行键，默认情况下，这个行键的返回是前闭后开区间，即包含起始行，单不包含中指行，如果我们想要同时包含起始行和终止行，那么我们可以使用此过滤器
	ColumnCountGetFilter	返回每行最多返回多少列，并在遇到一行的列数超过我们所设置的限制值的时候，结束扫描操作，使用在Scan上不合适，用在Get上用于限制返回的列数
	ColumnPrefixFilter	按照列名的前缀来筛选单元格的，如果我们想要对返回的列的前缀加以限制的话，可以使用这个过滤器
	ColumnPaginationFilter	它用来过滤返回的列，从offset开始的limit个列，此类也可以用来分页
	TimestampsFilter	
	RandomRowFilter	从名字上就可以看出其大概的用法，本过滤器的作用就是按照一定的几率（<=0会过滤掉所有的行，>=1会包含所有的行）来返回随机的结果集，对于同样的数据集，多次使用同一个RandomRowFilter会返回不通的结果集，对于需要随机抽取一部分数据的应用场景，可以使用此过滤器
Decorating Filters	SkipFilter	这是一种附加过滤器，其与ValueFilter结合使用，如果发现一行中的某一列不符合条件，那么整行就会被过滤掉
	WhileMatchFilter	这个过滤器的应用场景也很简单，如果你想要在遇到某种条件数据之前的数据时，就可以使用这个过滤器；当遇到不符合设定条件的数据的时候，整个扫描也就结束了
FilterList	FilterList	用于综合使用多个过滤器。其有两种关系：FilterList.Operator.MUST_PASS_ONE和FilterList.Operator.MUST_PASS_ALL，默认的是FilterList.Operator.MUST_PASS_ALL，顾名思义，它们分别是AND和OR的关系，并且FilterList可以嵌套使用FilterList，使我们能够表达更多的需求

HBase Shell 常用操作：DML– 自定义Filter

HBase Shell 常用操作：工具类

工具名	功能
assign	分配一个region，请谨慎使用。assign 't1,,1389754486055.3f6aacebea6c8b2b66ac7d1d565a358d.'
balance_switch	balance_switch true或者balance_switch flase，配置master是否执行平衡各个regionserver的region数量，当我们需要维护或者重启一个regionserver时，会 关闭balancer，这样就使得region在regionserver上的分布不均，这个时候需要手工的开启balance
balancer	执行均衡region的操作，如果成功发出指令将返回true，否则返回false，比如处于rit状态时无法做均衡
close_region	关闭region，可以直接让rs操作close_region '3f6aacebea6c8b2b66ac7d1d565a358d', 'node104,60020,1390292741528'，也可以让master操作close_region 't1,,1389754486055.3f6aacebea6c8b2b66ac7d1d565a358d.'
compact	执行紧缩操作，可以表，region或列族粒度compact 't1'，compact 'r1'，compact 't1', 'c1'，compact 'r1', 'c1'
flush	将memstore中数据刷入hdfs，可以表，也可以regionflush 'TABLENAME'，flush 'REGIONNAME‘ flush 'testtable'，将所有memstore刷新到hdfs，通常如果发现regionserver的内存使用过大，造成该机的 regionserver很多线程block，可以执行一下flush操作，这个操作会造成hbase的storefile数量剧增，应尽量避免这个操 作，还有一种情况，在hbase进行迁移的时候，如果选择拷贝文件方式，可以先停写入，然后flush所有表，拷贝文件
hlog_roll	对wal进行轮转，需传入rs完整名字hlog_roll 'node104,60020,1390292741528'
major_compact	Major compact，区别只是minor和major之间的差异，前者只是文件的合并，后者还会对过期数据，删除数据进行清理 通常生产环境会关闭自动major_compact(配置文件中hbase.hregion.majorcompaction设 为0)，选择一个晚上用户少的时间窗口手工major_compact
move	移动region被托管的地方，可以指定rs，也可以任由其随机选move 'ENCODED_REGIONNAME'，move 'ENCODED_REGIONNAME', 'SERVER_NAME'
split	拆分region，可以传入表名，可以传入region全名，可以指定分割点split 'tableName'，split 'tableName', 'splitKey'
unassign	将现有region下线，同时在新的地方打开，可以传入true强制执行unassign 't1,,1389754486055.3f6aacebea6c8b2b66ac7d1d565a358d.', true
zk_dump	将zk目前的信息dump出来

HBase Shell 常用操作：工具类：移动region，开启/关闭region

移动region

```
# 语法：move 'encodeRegionName', 'ServerName'
# encodeRegionName指的regioName后面的编码，ServerName指的是master-status的Region Servers列表
# 示例
hbase(main)>move '4343995a58be8e5bbc739af1e91cd72d', 'db-41.xxx.xxx.org,60020,1390274516739'
```

开启/关闭region

```
# 语法：balance_switch true|false
hbase(main)> balance_switch
```

HBase Shell 常用操作：工具类：split，major compaction

手动split

```
# 语法: split 'regionName', 'splitKey'
```

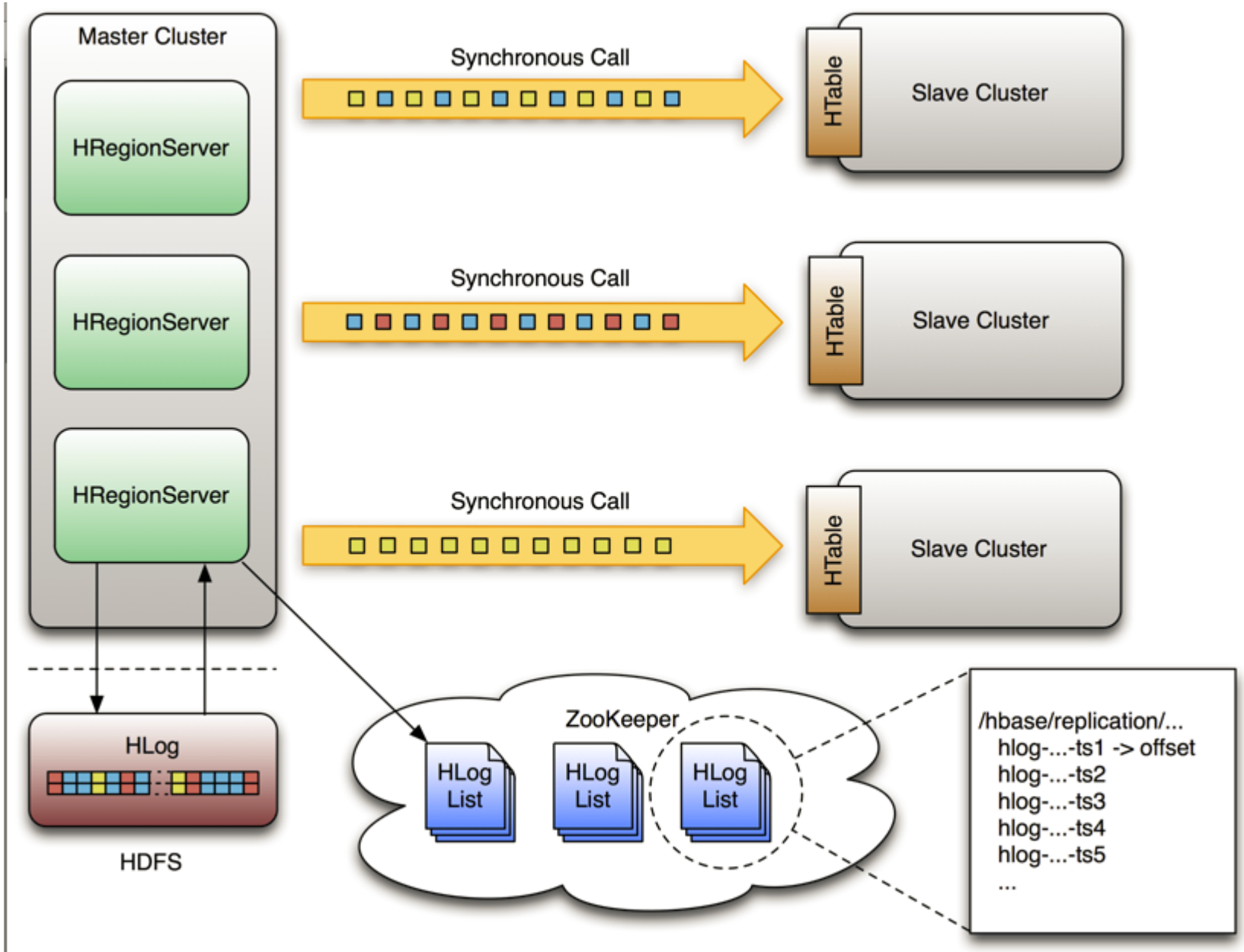
手动触发major compaction

```
#语法:
#Compact all regions in a table:
#hbase> major_compact 't1'
#Compact an entire region:
#hbase> major_compact 'r1'
#Compact a single column family within a region:
#hbase> major_compact 'r1', 'c1'
#Compact a single column family within a table:
#hbase> major_compact 't1', 'c1'
```

HBase Shell 常用操作：备份复制—原理

hbase的replication机制很像mysql statement-based replication。它是通过WALEdit和hlog来实现的。当请求发送给master cluster时，hlog日志放入hdfs的同时进入replication队列，由slave cluster通过zookeeper获取并写入slave的表中。目前的版本仅支持一个slave cluster

1 需要保证主从cluster上有相同的table，并且结构一致，都enable
2 保证主从cluster的版本都在0.90.0以上
3 主从cluster的机器是两两互通的
4 master cluster的hbase-site.xml中需要添加以下选项hbase.replication为true



HBase Shell 常用操作：备份复制

add_peer:增加一个复制集群

```
hbase> add_peer '1', “server1.cie.com:2181:/hbase”
hbase> add_peer '2', “zk1,zk2,zk3:2182:/hbase-prod”
```

remove_peer:减少复制到一个集群

```
hbase> remove_peer '1'
```

list_peers：罗列所有复制集群

```
hbase> list_peers
```

enable_peer：让某个集群开始复制

```
hbase> enable_peer '1'
```

disable_peer：让某个集群停止复制

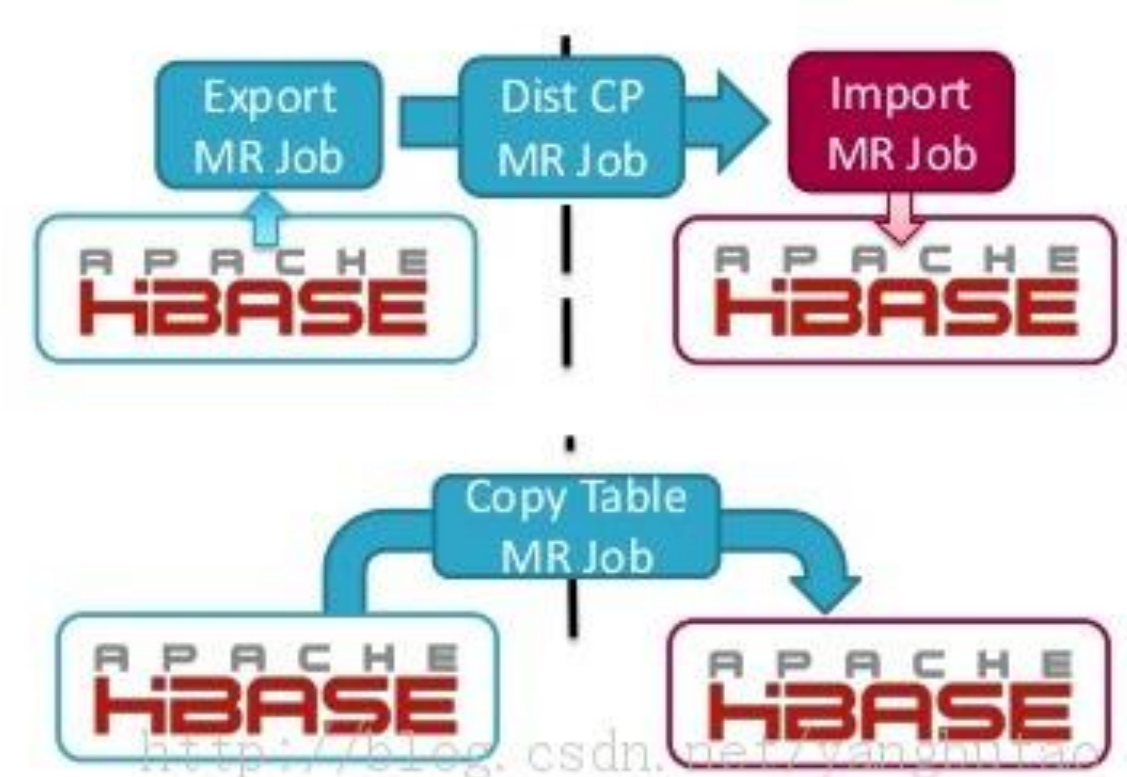
```
disable_peer '1'
```

start_replication：所有的复制集群都开始复制

stop_replication：所有的复制集群都停止复制

HBase Shell 常用操作：快照(黑历史)

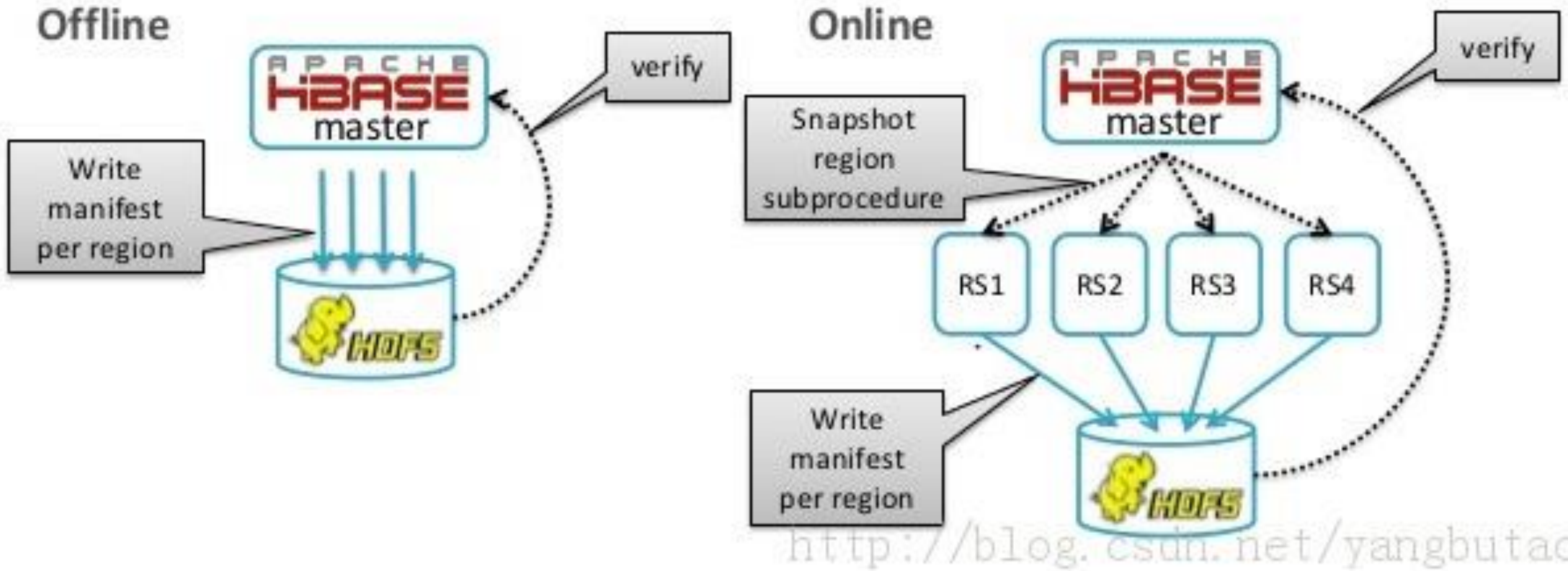
HBase以往数据的备份基于distcp或者copyTable等工具，这些备份机制或多或少对当前的online数据读写存在一定的影响



Snapshot提供了一种快速的数据备份方式，无需进行数据copy

快照就是一份元信息的合集，允许管理员恢复到表的先前状态。快照不是表的复制而是一个文件名称列表，因而不会复制数据。完全快照恢复是指恢复到之前的“表结构”以及当时的数据，快照之后发生的数据不会恢复

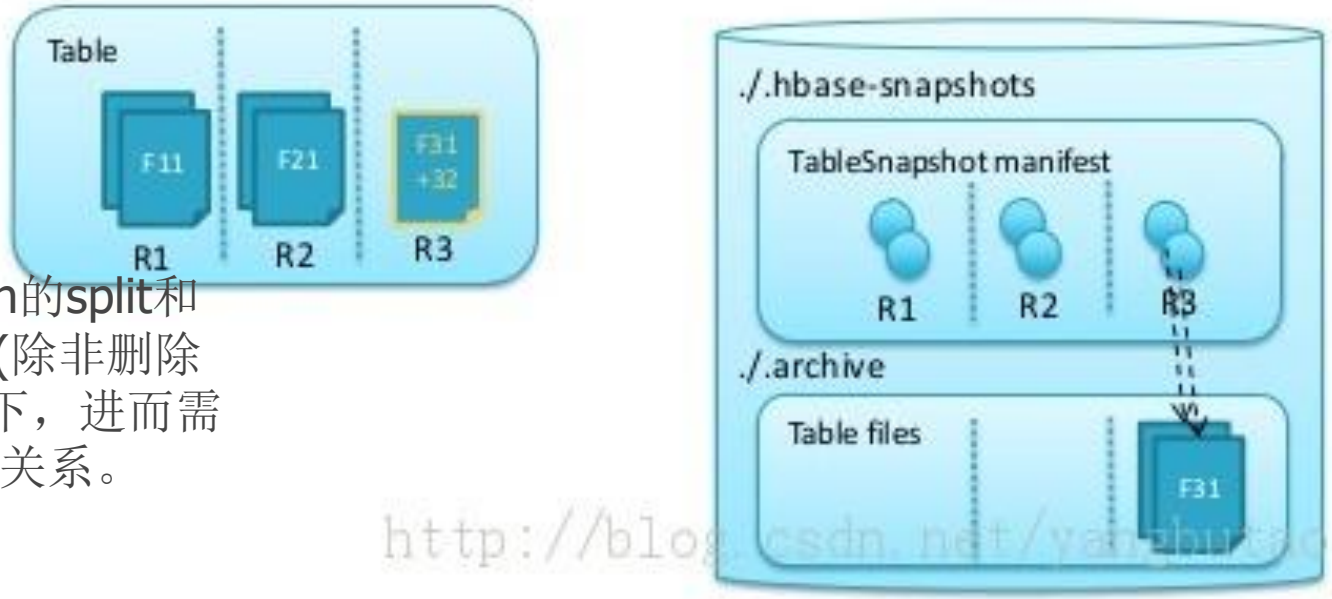
HBase Shell 常用操作：快照原理(离线/在线)



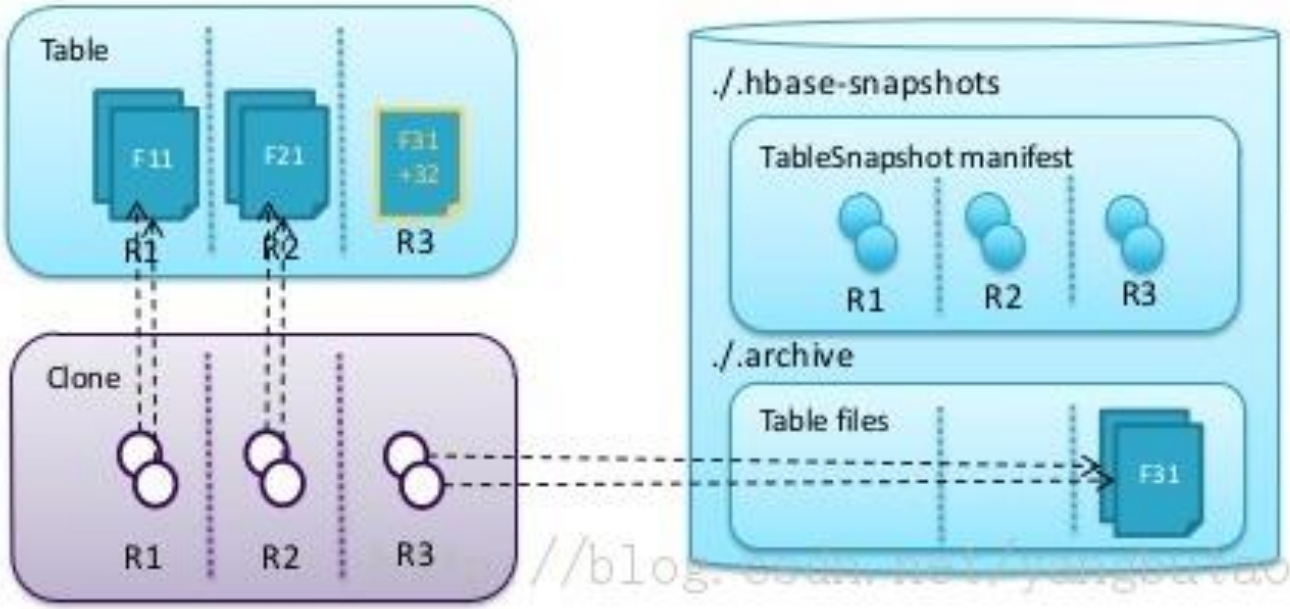
(1)离线方式是disabletable，由HBase Master遍历HDFS中的table metadata和hfiles，建立对他们的引用。
(2)在线方式是enabletable，由Master指示region server进行snapshot操作，在此过程中，master和regionserver之间类似两阶段commit的snapshot操作。

HBase Shell 常用操作：快照原理

HFile是不可变的，只能append和delete， region的split和compact，都不会对snapshot引用的文件做删除(除非删除snapshot文件)，这些文件会归档到archive目录下，进而需要重新调整snapshot文件中相关hfile的引用位置关系。



基于snapshot文件，可以做clone一个新表，restore，export到另外一个集群中操作；其中clone生成的新表只是增加元数据，相关的数据文件还是复用snapshot指定的数据文件



HBase Shell 常用操作：快照命令

snapshot 建立快照：
hbase> snapshot 'myTable','myTableSnapshot-122112'
list_snapshots 列出当前所有得快照：
hbase> list_snapshots
delete_snapshot 删除快照信息：
hbase> delete_snapshot'myTableSnapshot-122112'
clone_snapshot 基于快照，**clone**一个新表：
hbase> clone_snapshot'myTableSnapshot-122112', 'myNewTestTable'
restore_snapshot 基于快照恢复表：
hbase> disable 'myTable'
hbase> restore_snapshot'myTableSnapshot-122112'
导出到另外一个集群中：
$bin/hbase class org.apache.hadoop.hbase.snapshot.tool.ExportSnapshot -snapshotMySnapshot -copy-to hdfs:///srv2:8082/hbase -mappers 16

Hbase WebUI使用

http://dataunion.org/21730.html

https://www.zybuluo.com/xtccc/note/191871

Backup:HBase Compaction的前生今世

Compaction& Minor Compaction & Major Compaction

HBase是一种Log-Structured Merge Tree架构模式，用户数据写入先写WAL，再写缓存，满足一定条件后缓存数据会执行flush操作真正落盘，形成一个数据文件HFile。随着数据写入不断增多，flush次数也会不断增多，进而HFile数据文件就会越来越多。然而，太多数据文件会导致数据查询IO次数增多，因此HBase尝试着不断对这些文件进行合并，这个合并过程称为Compaction。

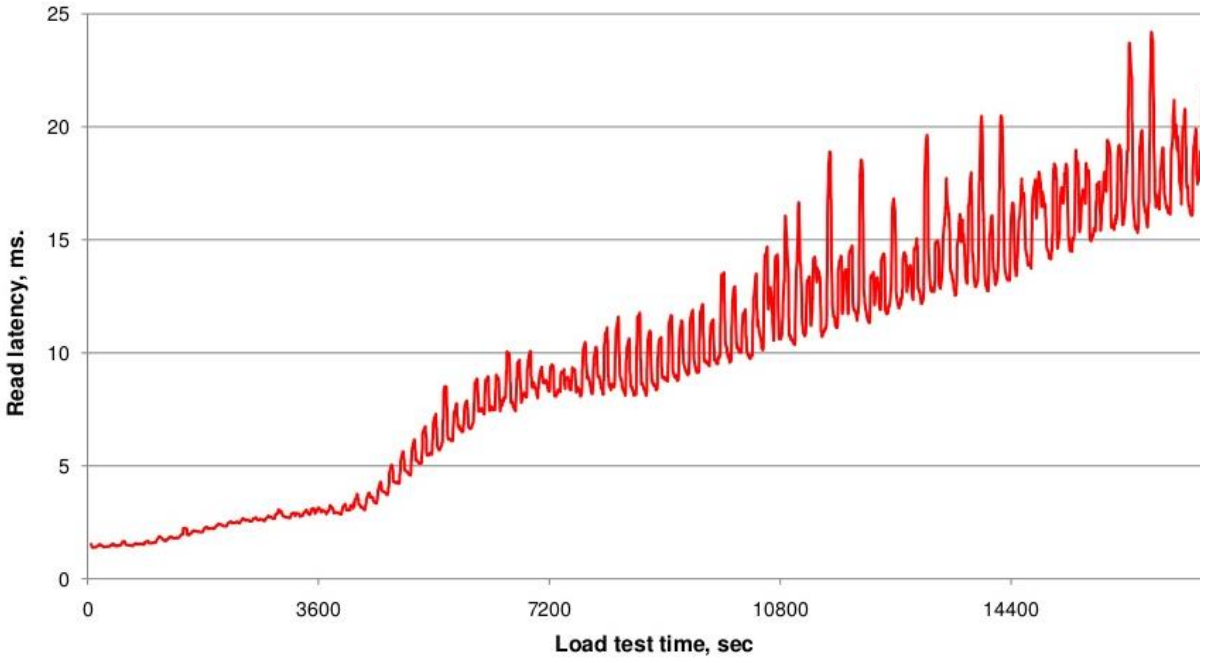
Compaction会从一个region的一个store中选择一些hfile文件进行合并。合并说来原理很简单，先从这些待合并的数据文件中读出KeyValues，再按照由小到大排列后写入一个新的文件中。之后，这个新生成的文件就会取代之前待合并的所有文件对外提供服务。HBase根据合并规模将Compaction分为了两类：MinorCompaction和MajorCompaction

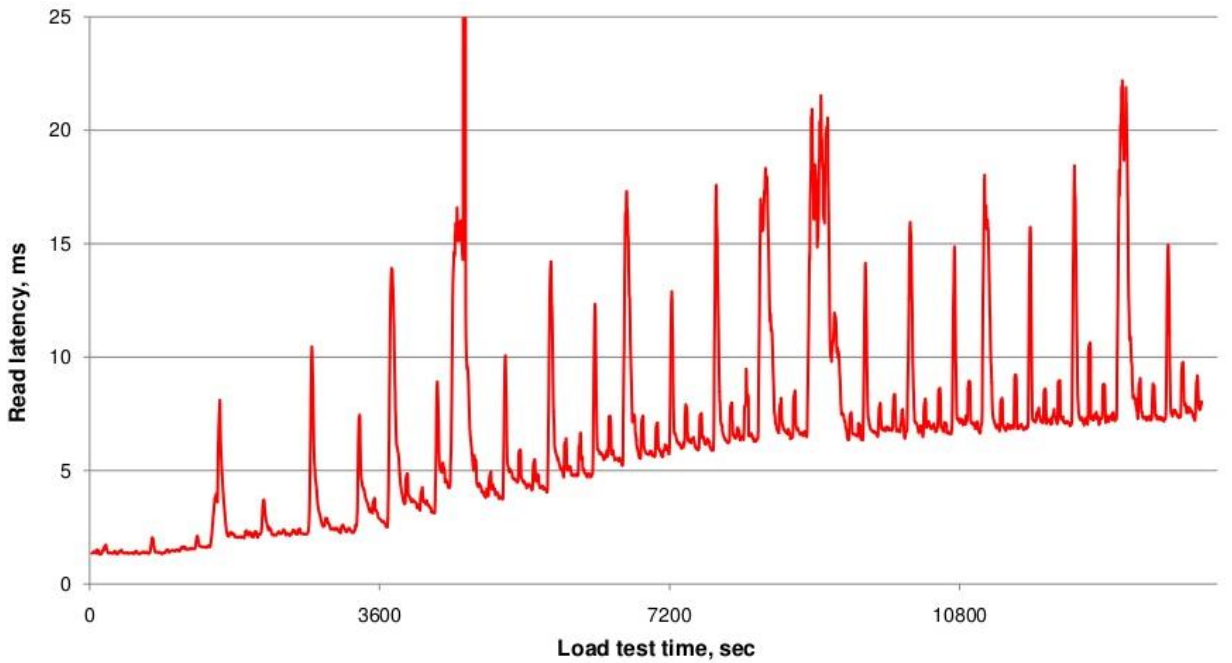
- Minor Compaction是指选取一些小的、相邻的StoreFile将他们合并成一个更大的StoreFile，在这个过程中不会处理已经Deleted或Expired的Cell。一次Minor Compaction的结果是更少并且更大的StoreFile。
- Major Compaction是指将所有的StoreFile合并成一个StoreFile，这个过程还会清理三类无意义数据：被删除的数据、TTL过期数据、版本号超过设定版本号的数据。另外，一般情况下，Major Compaction时间会持续比较长，整个过程会消耗大量系统资源，对上层业务有比较大的影响。因此线上业务都会将关闭自动触发Major Compaction功能，改为手动在业务低峰期触发。

Compaction作用 | 副作用

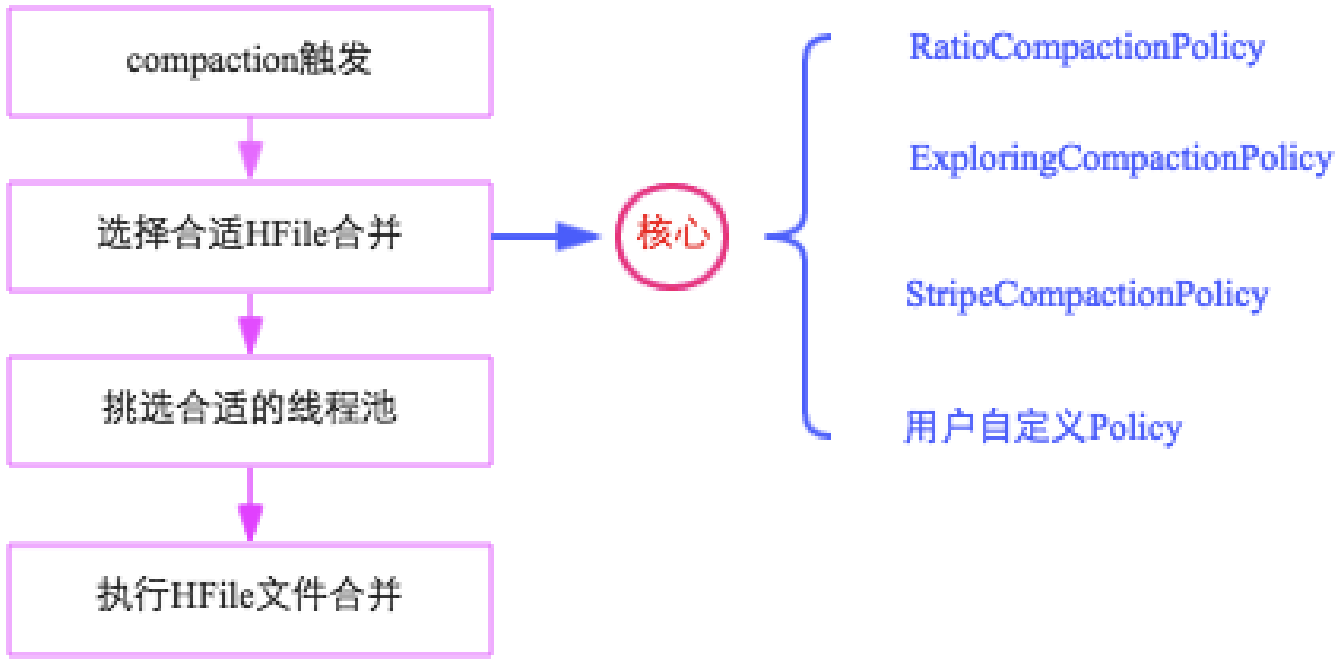
随着hfile文件数不断增多，一次查询就可能会需要越来越多的IO操作，延迟必然会越来越大，如下图一所示，随着数据写入不断增加，文件数不断增多，读取延时也在不断变大。而执行compaction会使得文件数基本稳定，进而IO Seek次数会比较稳定，延迟就会稳定在一定范围。然而，compaction操作重写文件会带来很大的带宽压力以及短时间IO压力。因此可以认为，Compaction就是使用短时间的IO消耗以及带宽消耗换取后续查询的低延迟。从图上来看，就是延迟有很大的毛刺，但总体趋势基本稳定不变，见下图二

Compaction会使得数据读取延迟一直比较平稳，但付出的代价是大量的读延迟毛刺和一定的写阻塞



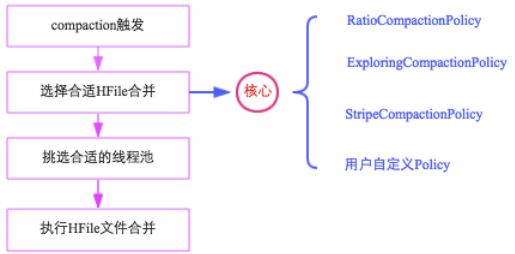


Compaction流程



整个Compaction始于特定的触发条件，比如flush操作、周期性地Compaction检查操作等。一旦触发，HBase会将该Compaction交由一个独立的线程处理，该线程首先会从对应store中选择合适的hfile文件进行合并，这一步是整个Compaction的核心，选取文件需要遵循很多条件，比如文件数不能太多、不能太少、文件大小不能太大等等，最理想的情况是，选取那些承载IO负载重、文件小的文件集，实际实现中，HBase提供了多个文件选取算法：RatioBasedCompactionPolicy、ExploringCompactionPolicy和StripeCompactionPolicy等，用户也可以通过特定接口实现自己的Compaction算法；选出待合并的文件后，HBase会根据这些hfile文件总大小挑选对应的线程池处理，最后对这些文件执行具体的合并操作

触发时机



最常见的因素有这么三种：Memstore Flush、后台线程周期性检查、手动触发

1. Memstore Flush: 应该说compaction操作的源头就来自flush操作，memstore flush会产生HFile文件，文件越来越多就需要compact。因此在每次执行完Flush操作之后，都会对当前Store中的文件数进行判断，一旦文件数＃ ＞，就会触发compaction。需要说明的是，compaction都是以Store为单位进行的，而在Flush触发条件下，整个Region的所有Store都会执行compact，所以会在短时间内执行多次compaction。

2. 后台线程周期性检查：后台线程CompactionChecker定期触发检查是否需要执行compaction，检查周期为：hbase.server.thread.wakefrequency*hbase.server.compactchecker.interval.multiplier。和flush不同的是，该线程优先检查文件数＃是否大于，一旦大于就会触发compaction。如果不满足，它会接着检查是否满足major compaction条件，简单来说，如果当前store中hfile的最早更新时间早于某个值mcTime，就会触发major compaction，HBase预想通过这种机制定期删除过期数据。上文mcTime是一个浮动值，浮动区间默认为［7-7*0.2，7+7*0.2］，其中7为hbase.hregion.majorcompaction，0.2为hbase.hregion.majorcompaction.jitter，可见默认在7天左右就会执行一次major compaction。用户如果想禁用major compaction，只需要将参数hbase.hregion.majorcompaction设为0

3. 手动触发：一般来讲，手动触发compaction通常是为了执行major compaction，原因有三，其一是因为很多业务担心自动major compaction影响读写性能，因此会选择低峰期手动触发；其二也有可能是用户在执行完alter操作之后希望立刻生效，执行手动触发major compaction；其三是HBase管理员发现硬盘容量不够的情况下手动触发major compaction删除大量过期数据；无论哪种触发动机，一旦手动触发，HBase会不做很多自动化检查，直接执行合并。

选择合适HFile合并

无论哪种选择策略，都会首先对该Store中所有HFile进行一一排查，排除不满足条件的部分文件

1. 排除当前正在执行compact的文件及其比这些文件更新的所有文件（SequenceId更大）
2. 排除某些过大的单个文件，如果文件大小大于hbase.hzstore.compaction.max.size（默认Long最大值），则被排除，否则会产生大量IO消耗

经过排除的文件称为候选文件，HBase接下来会再判断是否满足major compaction条件，如果满足，就会选择全部文件进行合并。判断条件有下面三条，只要满足其中一条就会执行major compaction：

1. 用户强制执行major compaction
2. 长时间没有进行compact（CompactionChecker的判断条件2）且候选文件数小于hbase.hstore.compaction.max（默认10）
3. Store中含有Reference文件，Reference文件是split region产生的临时文件，只是简单的引用文件，一般必须在compact过程中删除

如果不满足major compaction条件，就必然为minor compaction，HBase主要有两种minor策略：RatioBasedCompactionPolicy和ExploringCompactionPolicy

minor策略： RatioBasedCompactionPolicy

从老到新逐一扫描所有候选文件，满足其中条件之一便停止扫描：
（1）当前文件大小 < 比它更新的所有文件大小总和 * ratio，其中ratio是一个可变的比例，在高峰期时ratio为1.2，非高峰期为5，也就是非高峰期允许compact更大的文件。那什么时候是高峰期，什么时候是非高峰期呢？用户可以配置参数hbase.offpeak.start.hour和hbase.offpeak.end.hour来设置高峰期
（2）当前所剩候选文件数 <= hbase.store.compaction.min（默认为3）
停止扫描后，待合并文件就选择出来了，即为当前扫描文件+比它更新的所有文件

Ratio策略是0.94版本的默认策略

minor策略： ExploringCompactionPolicy

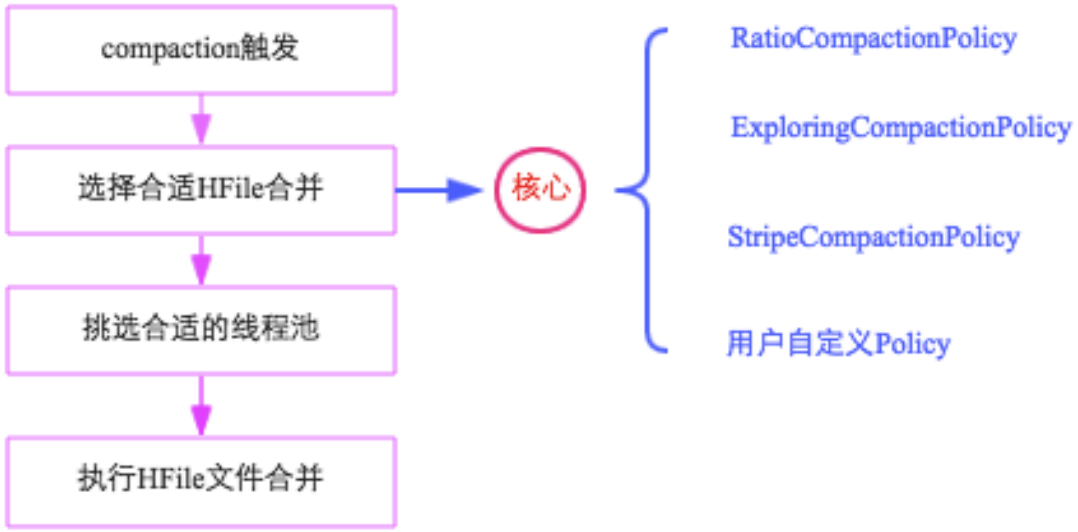
该策略思路基本和RatioBasedCompactionPolicy相同，不同的是，Ratio策略在找到一个合适的文件集合之后就停止扫描了，而Exploring策略会记录下所有合适的文件集合，并在这些文件集合中寻找最优解。最优解可以理解为：待合并文件数最多或者待合并文件数相同的情况下文件大小较小，这样有利于减少compaction带来的IO消耗

0.96版本之后默认策略就换为了Exploring策略

主要算法如下：

1.从头到尾遍历文件，判断所有符合条件的组合
2.选择组合的文件数必须 >= minFiles（默认值：3）
3.选择组合的文件数必须 <= maxFiles（默认值：10）
4.计算组合的文件总大小size，size必须 <= maxCompactSize(通过hbase.hstore.compaction.max.size配置，默认值：LONG.MAX_VALUE，相当于没起作用，官方文档里面说只有觉得compaction经常发生并且没有多大的用时，可以修改这个值)
5.组合的文件大小 < minCompactSize 则是符合要求，如果 >= minCompactSize ，还需要判断filesInRatio
6.filesInRatio算法：FileSize(i) <= (Sum(0,N,FileSize(_)) - FileSize(i)) * Ratio，也就是说组合里面的所有单个文件大小都必须满足 singleFileSize <= (totalFileSize - singleFileSize) * currentRatio，此算法的意义是为了限制太大的compaction，选择出来的文件不至于有一个很大的，应该尽可能先合并一些小的大小相差不大的文件

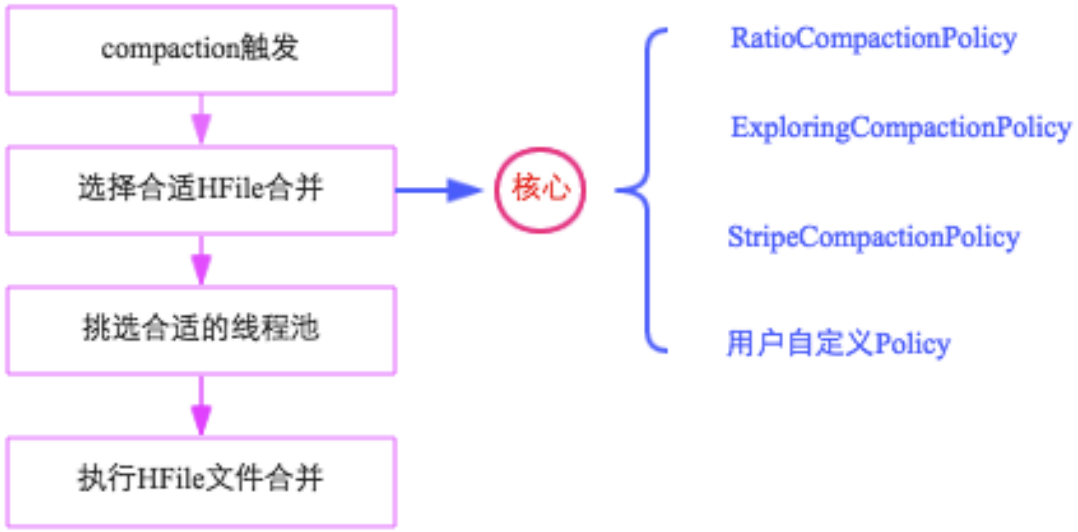
挑选合适的线程池



HBase实现中有一个专门的线程CompactSplitThead负责接收compact请求以及split请求，而且为了能够独立处理这些请求，这个线程内部构造了多个线程池：largeCompactions、smallCompactions以及splits等，其中splits线程池负责处理所有的split请求，largeCompactions和smallCompaction负责处理所有的compaction请求，其中前者用来处理大规模compaction，后者处理小规模compaction。这里需要明白三点：

1. 上述设计目的是为了能够将请求独立处理，提供系统的处理性能。
2. 哪些compaction应该分配给largeCompactions处理，哪些应该分配给smallCompactions处理？是不是Major Compaction就应该交给largeCompactions线程池处理？不对。这里有个分配原则：待compact的文件总大小如果大于值throttlePoint（可以通过参数hbase.hregion.majorcompaction配置，默认为2.5G），分配给largeCompactions处理，否则分配给smallCompactions处理。
3. largeCompactions线程池和smallCompactions线程池默认都只有一个线程，用户可以通过参数hbase.regionserver.thread.compaction.large和hbase.regionserver.thread.compaction.small进行配置

执行HFile文件合并



1. 分别读出待合并hfile文件的KV，并顺序写到位于./tmp目录下的临时文件中
2. 将临时文件移动到对应region的数据目录
3. 将compaction的输入文件路径和输出文件路径封装为KV写入WAL日志，并打上compaction标记，最后强制执行sync
4. 将对应region数据目录下的compaction输入文件全部删除

上述四个步骤看起来简单，但实际是很严谨的，具有很强的容错性和完美的幂等性：

1. 如果RS在步骤2之前发生异常，本次compaction会被认为失败，如果继续进行同样的compaction，上次异常对接下来的compaction不会有任何影响，也不会对读写有任何影响。唯一的影响就是多了一份多余的数据。
2. 如果RS在步骤2之后、步骤3之前发生异常，同样的，仅仅会多一份冗余数据。
3. 如果在步骤3之后、步骤4之前发生异常，RS在重新打开region之后首先会从WAL中看到标有compaction的日志，因为此时输入文件和输出文件已经持久化到HDFS，因此只需要根据WAL移除掉compaction输入文件即可

Compaction的前生今世－改造之路

HBase对于compaction的设计总是会追求一个平衡点，一方面需要保证compaction的基本效果，另一方面又不会带来严重的IO压力。然而，并没有一种设计策略能够适用于所有应用场景或所有数据集。

HBase就希望能够提供一种机制可以在不同业务场景下针对不同设计策略进行测试，另一方面也可以让用户针对自己的业务场景选择合适的compaction策略

一方面提供了Compaction插件接口，用户只需要实现这些特定的接口

另一方面，0.96版本之后Compaction可以支持table/cf粒度的策略设置，使得用户可以根据应用场景为不同表/列族选择不同的compaction策略

```
alter ’table1’ , CONFIGURATION => {‘hbase.store.engine.class’ =>
‘org.apache.hadoop.hbase.regionserver.StripStoreEngine’, … }
```

优化compaction的共性特征

1. 减少参与compaction的文件数：这个很好理解，实现起来却比较麻烦，首先需要将文件根据rowkey、version或其他属性进行分割，再根据这些属性挑选部分重要的文件参与合并；另一方面，尽量不要合并那些大文件，减少参与合并的文件数。
2. 不要合并那些不需要合并的文件：比如OpenTSDB应用场景下的老数据，这些数据基本不会查询到，因此不进行合并也不会影响查询性能
3. 小region更有利于compaction：大region会生成大量文件，不利于compaction；相反，小region只会生成少量文件，这些文件合并不会引起很大的IO放大

FIFO Compaction

Tire-Based Compaction

Stripe Compaction

Level Compaction

FIFO Compaction

它会选择那些过期的数据文件，即该文件内所有数据都已经过期。因此，对应业务的列族必须设置TTL，否则肯定不适合该策略。需要注意的是，该策略只做这么一件事情：收集所有已经过期的文件并删除。这样的应用场景主要包括：

1. 大量短时间存储的原始数据，比如推荐业务，上层业务只需要最近时间内用户的行为特征，利用这些行为特征进行聚合为用户进行推荐。再比如Nginx日志，用户只需要存储最近几天的日志，方便查询某个用户最近一段时间的操作行为等等
2. 所有数据能够全部加载到block cache（RAM/SSD），假如HBase有1T大小的SSD作为block cache，理论上就完全不需要做合并，因为所有读操作都是内存操作。

因为FIFO Compaction只是收集所有过期的数据文件并删除，并没有真正执行重写（几个小文件合并成大文件），因此不会消耗任何CPU和IO资源，也不会从block cache中淘汰任何热点数据。所以，无论对于读还是写，该策略都会提升吞吐量、降低延迟。

开启FIFO Compaction（表设置&列族设置）

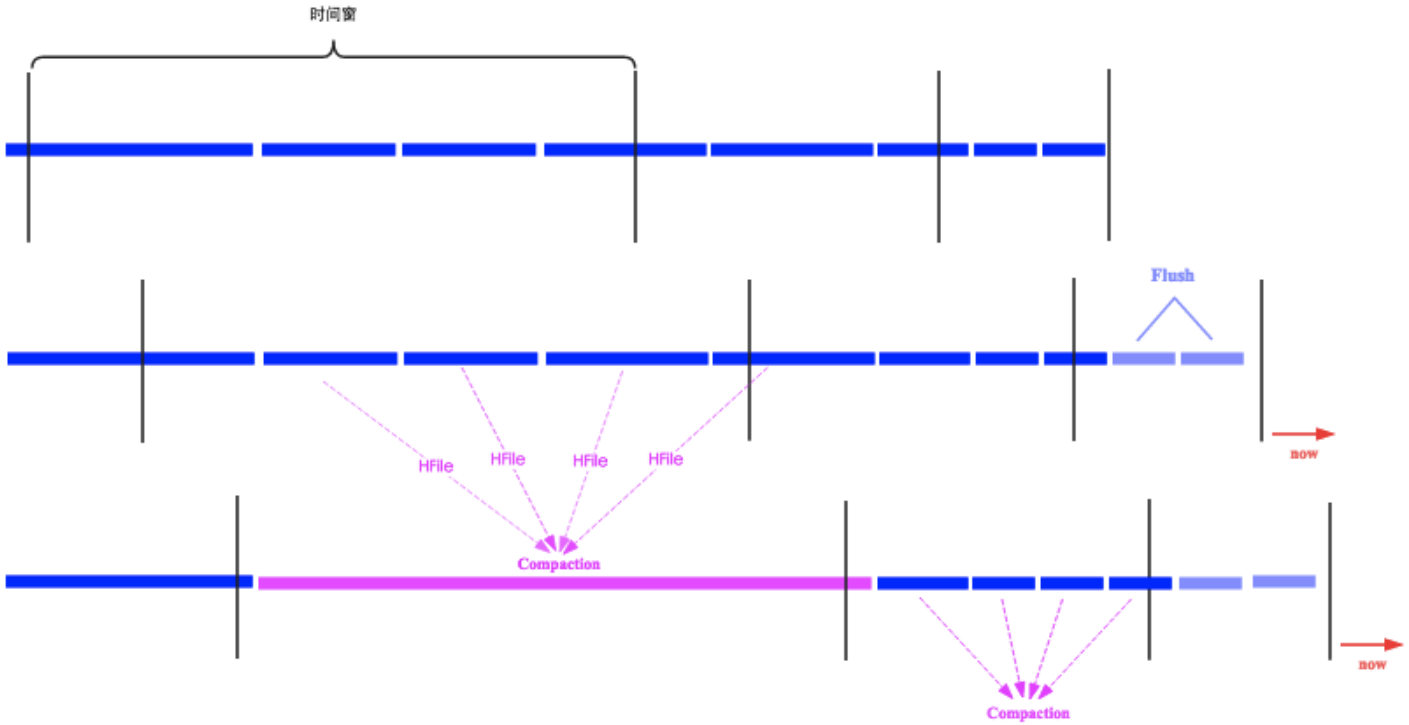
```
HTableDescriptor desc = new HTableDescriptor(tableName);
        desc.setConfiguration(DefaultStoreEngine.DEFAULT_COMPACTION_POLICY_CLASS_KEY,
        FIFOCompactionPolicy.class.getName());
```

```
HColumnDescriptor desc = new HColumnDescriptor(family);
          desc.setConfiguration(DefaultStoreEngine.DEFAULT_COMPACTION_POLICY_CLASS_KEY,
          FIFOCompactionPolicy.class.getName());
```

Tire-Based Compaction

现实业务中，有很大比例的业务都存在明显的热点数据，而其中最常见的情况是：最近写入到的数据总是最有可能被访问到，而老数据被访问到的频率就相对比较低。这种方案会根据候选文件的新老程度将其分为多个不同的等级，每个等级都有对应等级的参数，比如参数Compation Ratio，表示该等级文件选择时的选择几率，Ratio越大，该等级的文件越有可能被选中参与Compaction。而等级数、每个等级参数都可以通过CF属性在线更新。HBase计划在2.0.0版本发布基于时间划分等级的实现方式－Date Tierd Compaction Policy

HBase更多地参考了Cassendra的实现方案：基于时间窗的时间概念。如下图所示，时间窗的大小可以进行配置，其中参数base_time_seconds代表初始化时间窗的大小，默认为1h，表示最近一小时内flush的文件数据都会落入这个时间窗内，所有想读到最近一小时数据请求只需要读取这个时间窗内的文件即可。后面的时间窗窗口会越来越大，另一个参数max_age_days表示比其更老的文件不会参与compaction。



1. 特别适合使用的场景：时间序列数据，默认使用TTL删除。类似于“获取最近一小时／三小时／一天”场景，同时不会执行delete操作。最典型的例子就是基于Open-TSDB的监控系统

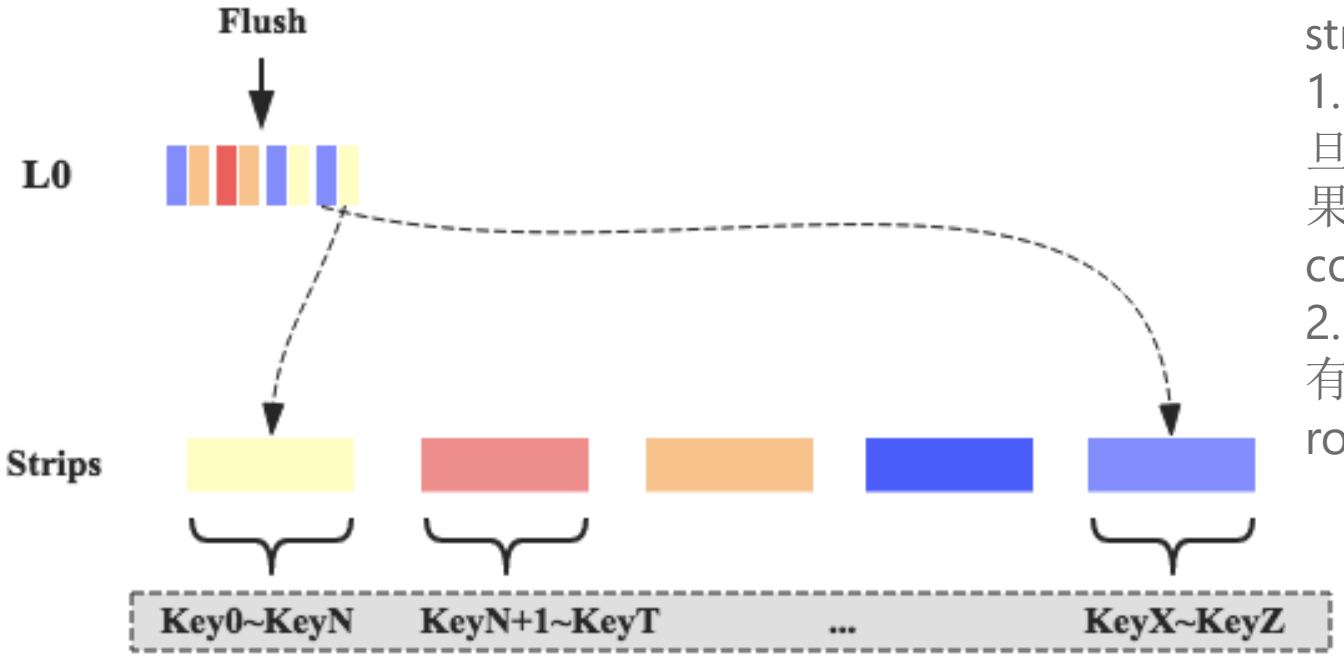
2. 比较适合的应用场景：时间序列数据，但是会有全局数据的更新操作以及少部分的删除操作。

3. 不适合的应用场景：非时间序列数据，或者大量的更新数据更新操作和删除操作。

Stripe Compaction

stripe compaction会将整个store中的文件按照Key划分为多个Range，在这里称为stripe，stripe的数量可以通过参数设定，相邻的stripe之间key不会重合。实际上在概念上来看这个stripe类似于sub-region的概念，即将一个大region切分成了很多小的sub-region。

memstore执行flush之后形成hfile，这些hfile并不会马上写入对应的stripe，而是放到一个称为L0的地方，用户可以配置L0可以放置hfile的数量。一旦L0放置的文件数超过设定值，系统就会将这些hfile写入对应的stripe：首先读出hfile的KVs，再根据KV的key定位到具体的stripe，将该KV插入对应stripe的文件中即可，如下图所示。之前说过stripe就是一个个小的region，所以在stripe内部，依然会像正常region一样执行minor compaction和major compaction，可以预想到，stripe内部的major compaction并不会太多消耗系统资源。另外，数据读取也很简单，系统可以根据对应的Key查找到对应的stripe，然后在stripe内部执行查找，因为stripe内数据量相对很小，所以也会一定程度上提升数据查找性能。



stripe compaction比较擅长的业务场景：

1. 大Region。小region没有必要切分为stripes，一旦切分，反而会带来额外的管理开销。一般默认如果region大小小于2G，就不适合使用stripe compaction。
2. RowKey具有统一格式，stripe compaction要求所有数据按照Key进行切分，切分为多个stripe。如果rowkey不具有统一格式的话，无法进行切分。

控制Compaction执行阶段的读写吞吐量

上述几种策略都是根据不同的业务场景设置对应的文件选择策略，核心都是减少参与compaction的文件数，缩短整个compaction执行的时间，间接降低compaction的IO放大效应，减少对业务读写的延迟影响。然而，如果不对Compaction执行阶段的读写吞吐量进行限制的话也会引起短时间大量系统资源消耗，影响用户业务延迟。HBase社区也意识到了这个问题，也提出了一定的应对策略

Limit Compaction Speed

Compaction BandWidth Limit

Limit Compaction Speed

该优化方案通过感知Compaction的压力情况自动调节系统的Compaction吞吐量，在压力大的时候降低合并吞吐量，压力小的时候增加合并吞吐量。基本原理为：

1. 在正常情况下，用户需要设置吞吐量下限参数“hbase.hstore.compaction.throughput.lower.bound”(默认10MB/sec) 和上限参数“hbase.hstore.compaction.throughput.higher.bound”(默认20MB/sec)，而hbase实际会工作在吞吐量为lower + (higer – lower) * ratio的情况下，其中ratio是一个取值范围在0到1的小数，它由当前store中待参与compation的file数量决定，数量越多，ratio越小，反之越大。
2. 如果当前store中hfile的数量太多，并且超过了参数blockingFileCount，此时所有写请求就会阻塞等待compaction完成，这种场景下上述限制会自动失效。

Compaction BandWidth Limit

在某些情况下Compaction还会因为大量消耗带宽资源从而严重影响其他业务。为什么Compaction会大量消耗带宽资源呢？主要有两点原因：
1. 正常请求下，compaction尤其是major compaction会将大量数据文件合并为一个大HFile，读出所有数据文件的KVs，然后重新排序之后写入另一个新建的文件。如果待合并文件都在本地，那么读就是本地读，不会出现垮网络的情况。但是因为数据文件都是三副本，因此写的时候就会垮网络执行，必然会消耗带宽资源。
2. 原因1的前提是所有待合并文件都在本地的情况，那在有些场景下待合并文件有可能并不全在本地，即本地化率没有达到100%，比如执行过balance之后就会有很多文件并不在本地。这种情况下读文件的时候就会垮网络读，如果是major compaction，必然也会大量消耗带宽资源。
可以看出来，垮网络读是可以通过一定优化避免的，而垮网络写却是不可能避免的。因此优化Compaction带宽消耗，一方面需要提升本地化率（一个优化专题，在此不详细说明），减少垮网络读；另一方面，虽然垮网络写不可避免，但也可以通过控制手段使得资源消耗控制在一个限定范围，HBase在这方面也参考fb也做了一些工作：

Compaction BandWidth Limit

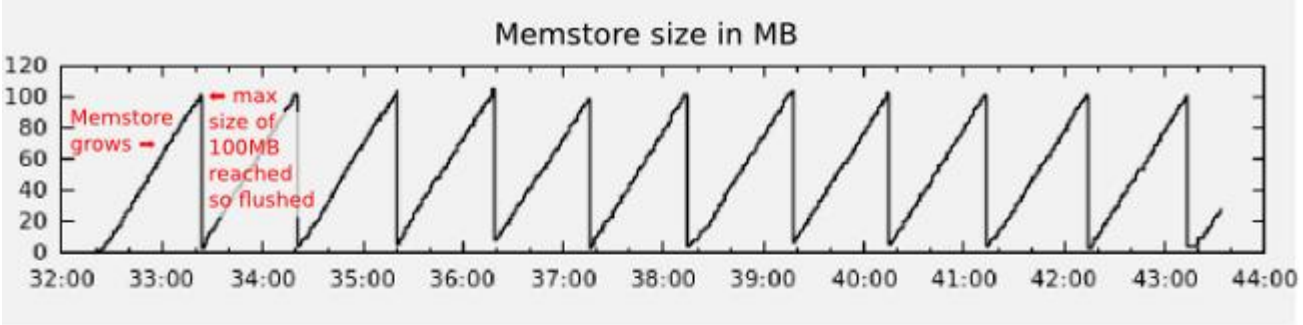
原理其实和Limit Compaction Speed思路基本一致，它主要涉及两个参数：compactBwLimit和numOfFilesDisableCompactLimit，作用分别如下：
1. compactBwLimit：一次compaction的最大带宽使用量，如果compaction所使用的带宽高于该值，就会强制令其sleep一段时间
2. numOfFilesDisableCompactLimit：很显然，在写请求非常大的情况下，限制compaction带宽的使用量必然会导致HFile堆积，进而会影响到读请求响应延时。因此该值意义就很明显，一旦store中hfile数量超过该设定值，带宽限制就会失效。

VISUALIZING HBASE FLUSHES AND COMPACTIONS

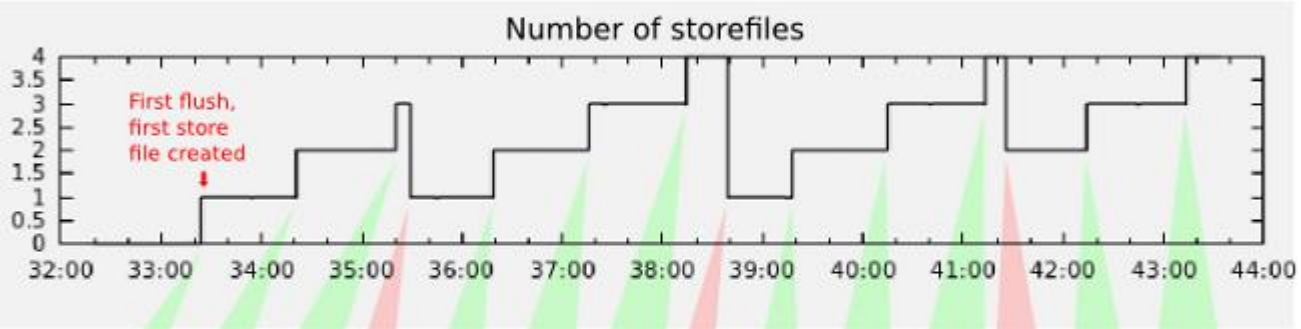
http://outerthought.org/blog/465-ot/version/-1

场景1：不停的写入

in a loop we perform a put of a row with a single 1k value. The table has one column family and only one region. The flush size size of the memstore has been set to 100MB (hbase.hregion.memstore.flush.size), the maximum file size (hbase.hregion.max.filesize) has been set high enough to avoid splits from happening. All tests in this blog have been done on a single node (my laptop). The main goal here is to illustrate how the minor compaction process decides when and what to merge

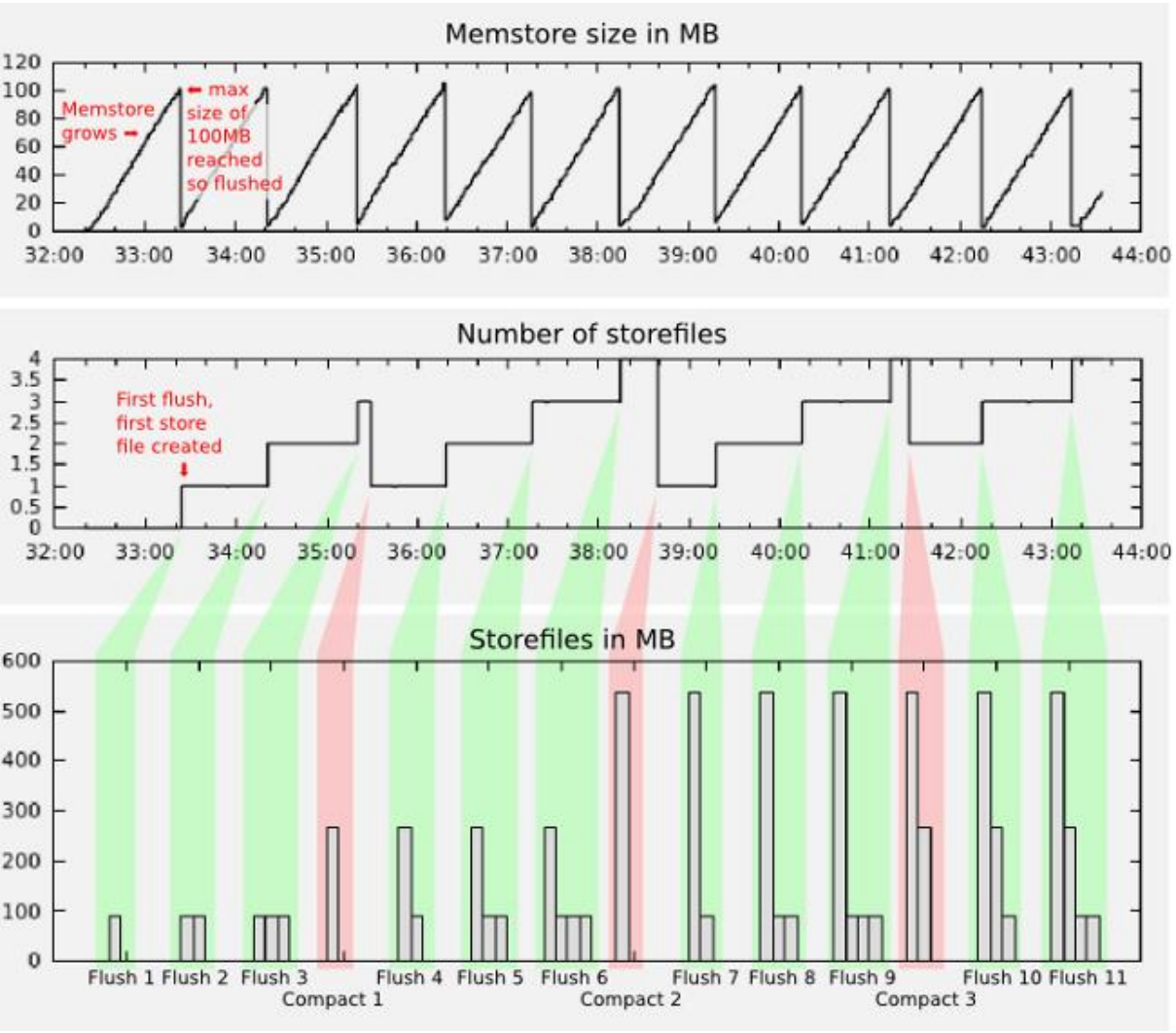


We see that the memstore grows gradually until it reaches its flush size of 100MB, when this happens it is flushed to disk in a new store file. This process repeats itself until we have 3 store files. At this point, a compaction is performed which merges the files together into one file.



The second compaction is only performed after 4 store files are written. How does HBase decide this?
The algorithm is basically as follows:
1.Run over the set of all store files, from oldest to youngest
2.If there are more than 3 (hbase.hstore.compactionThreshold) store files left and the current store file is 20% larger then the sum of all younger store files, and it is larger than the memstore flush size, then we go on to the next, younger, store file and repeat step 2.
3.Once one of the conditions in step two is not valid anymore, the store files from the current one to the youngest one are the ones that will be merged together. If there are less than the compactionThreshold, no merge will be performed. There is also a limit which prevents more than 10 (hbase.hstore.compaction.max) store files to be merged in one compaction.

场景1：不停的写入



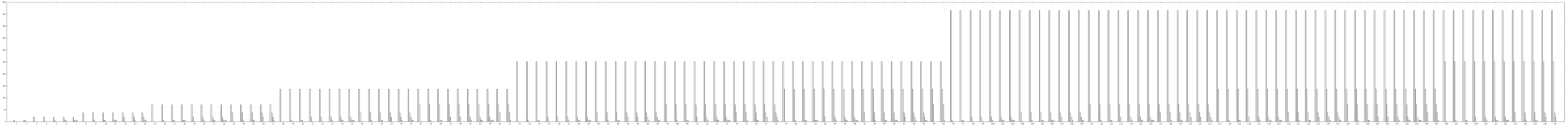
Now, with this knowledge, we can see why the second compaction does not run after flush 5: the oldest file was about ~240MB, the two younger ones together ~160MB, 240 > 160 * 1.2

When the second compaction runs after flush 6, it does merge all the files together, instead of just the 3 youngest, since ~240 > 3*80*1.2 is not true. After flush 9, we have ~480 > 3*80*1.2, so the oldest store file is not included in the merge.

With the default configuration of HBase, if a store file would be 480MB, the region will be split. I have on purpose augmented the limit to illustrate what happens if there are more store files, though I could also have lowered the memstore flush size.

If you wonder how the store file merging behaves over longer time, below is the result of a longer run with 157 flushes of 10 MB

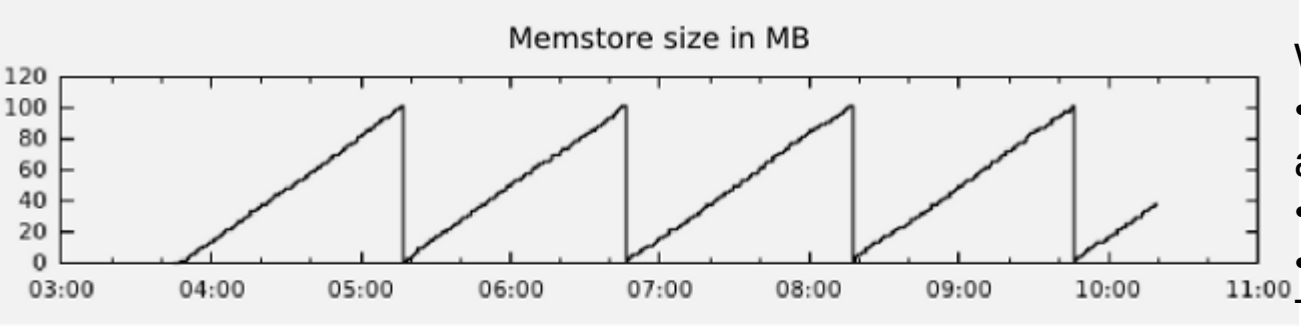
场景1：不停的写入

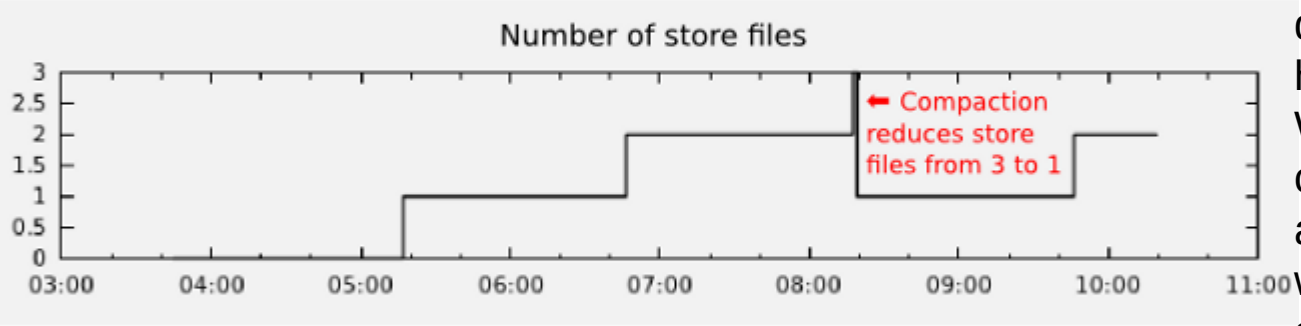


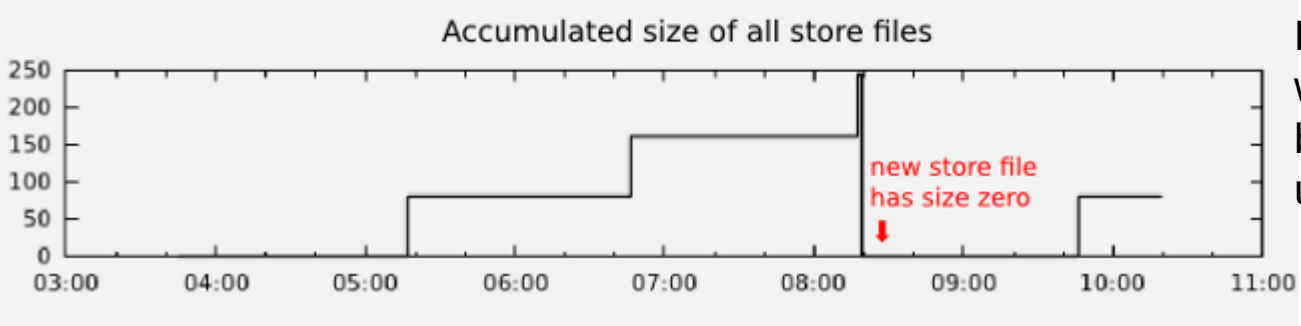
The largest number of store files at any time is 7 (see number 143 in the chart), though the oldest (biggest) one is only merged occasionally. It is a good thing that the number of store files stays small, because when you read a row from HBase it needs to check all the store files and the memstore (this can be optimized through **bloomfilters** or by specifying**timestamp ranges**). The number 7 might make you think of the property hbase.hstore.blockingStoreFiles, which has a default value of 7, but I augmented it for this test to be sure it did not have any influence.

场景二：有删除操作情况下的compaction

When you perform a delete in HBase, nothing gets deleted immediately, rather a delete marker (a.k.a. thombstone) is written. This is because HBase does not modify files once they are written.
The deletes are processed during the major compaction process, at which point the data they hide and the delete marker itself will not be present in the merged file.
Lets illustrate this with an extreme scenario. The following graphs are from a scenario where, in a loop, we put a row and then immediately delete the row. So conceptually, if we look at the table during the test, it will contain at most one row. Nevertheless, the store files will only grow.







What we see is:
•the memstore grows and flushes three times, three store files are created
•after the third store file is written, a compaction is performed
•the result of the compaction is a new store file of size 0.
This is a surprising result: only major compactions process deletes, and major compactions supposedly only run every 24 hours.
What happens is that if the set of store files selected for compaction is the set of all store files, then HBase decides to do a major compaction. In fact, it should, otherwise the 24H check, which looks at the time of the oldest store file, would mistakenly assume that this one file was the result of a major compaction.
Besides explicit deletes, there are other cases where store files will shrink after a major compaction: cells that are removed because there are more than max-versions versions of it, or when using the TTL feature.

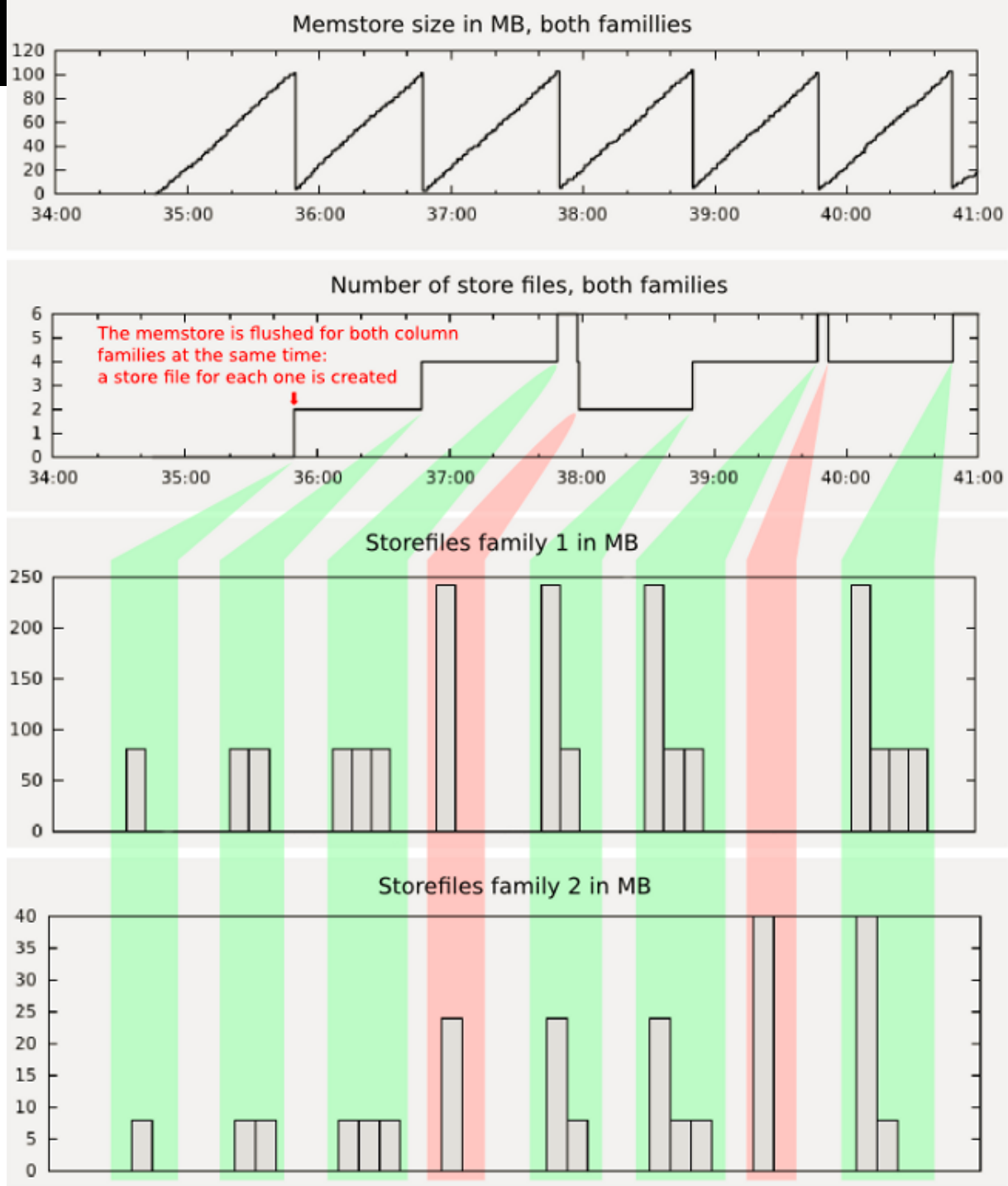
场景三： Multiple column families

Each column family has its own memstore and its own set of store files. But all the column families within a table stick together. They will all flush their memstore at the same time (this might change). They are also splitted in the same regions, so when one of the column families grows to large, the other ones, how small they might be, will also split.

The illustration below shows a scenario where a table with two column families is used. In family 2, we only do a put for every 10 puts we do in family 1. So family 1 will grow much faster than family 2 (family 2 is sparse family).

The memstore and storefile count graphs show the data for both families counted together. Note how at each memstore flush, the number of store files increases with two.

You can also see how a second compaction is performed for family 2 but not for family 1. This is because all store files in family 2 are smaller than the memstore flush size of 100 MB.

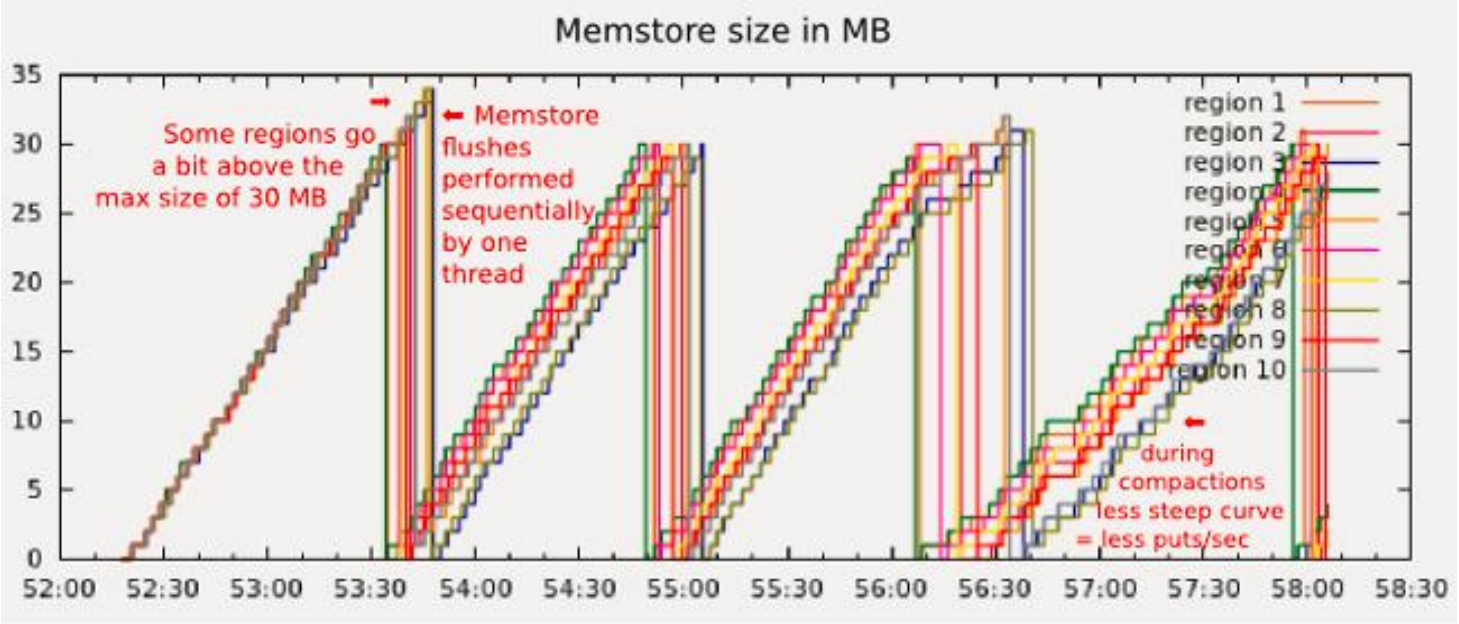


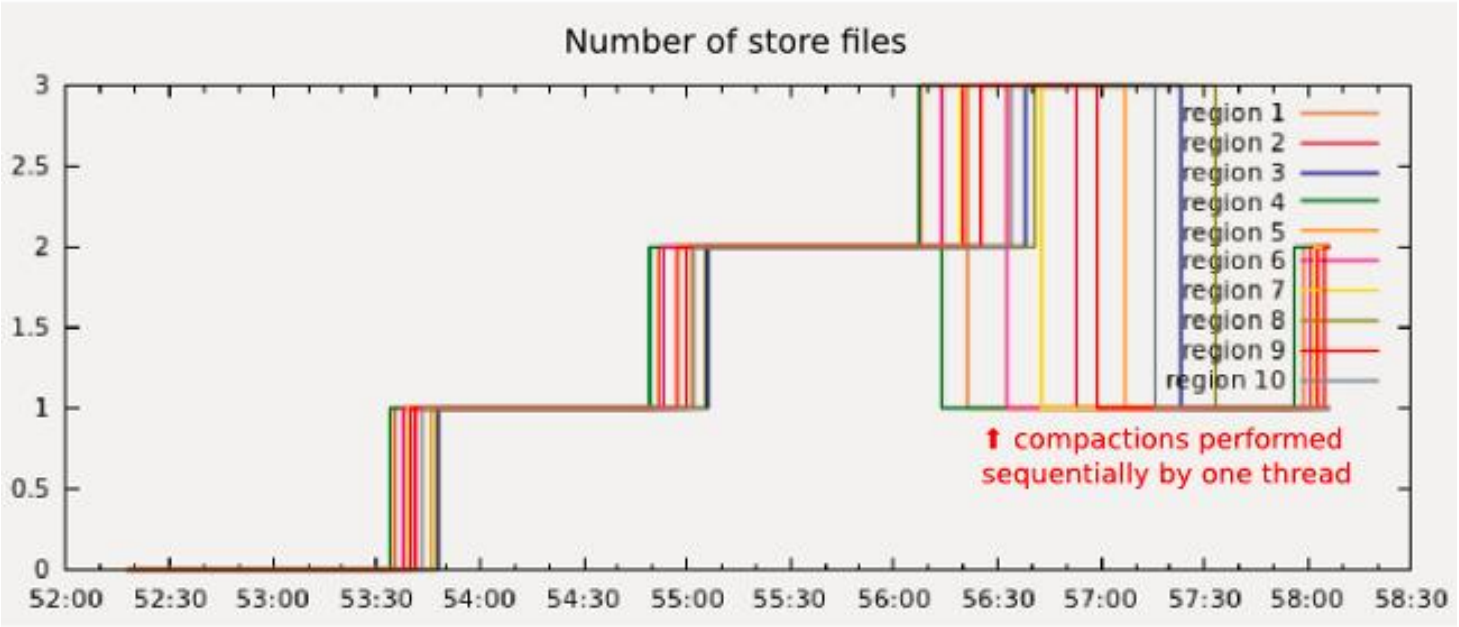
场景四： Multiple regions

For this test, a table was created with 10 initials regions, divided so that random UUIDs would be evenly spread over these regions. This time the memstore flush size was set to 30 MB.

As the puts are performed, the memstores of each region should fill up evenly, and so we expect them to all reach the flush size at about the same time, and thus to flush at about the same time. However, in the HBase region server there is only one thread which performs flushes, so they happen in sequence. This means that the memstores of some regions will keep growing above their limit. The property hbase.hregion.memstore.block.multiplier controls how many times a memstore can grow above its limit before HBase blocks updates. The default is 2, in our example here this means the memstore would be allowed to grow to 2*30=60MB maximum.

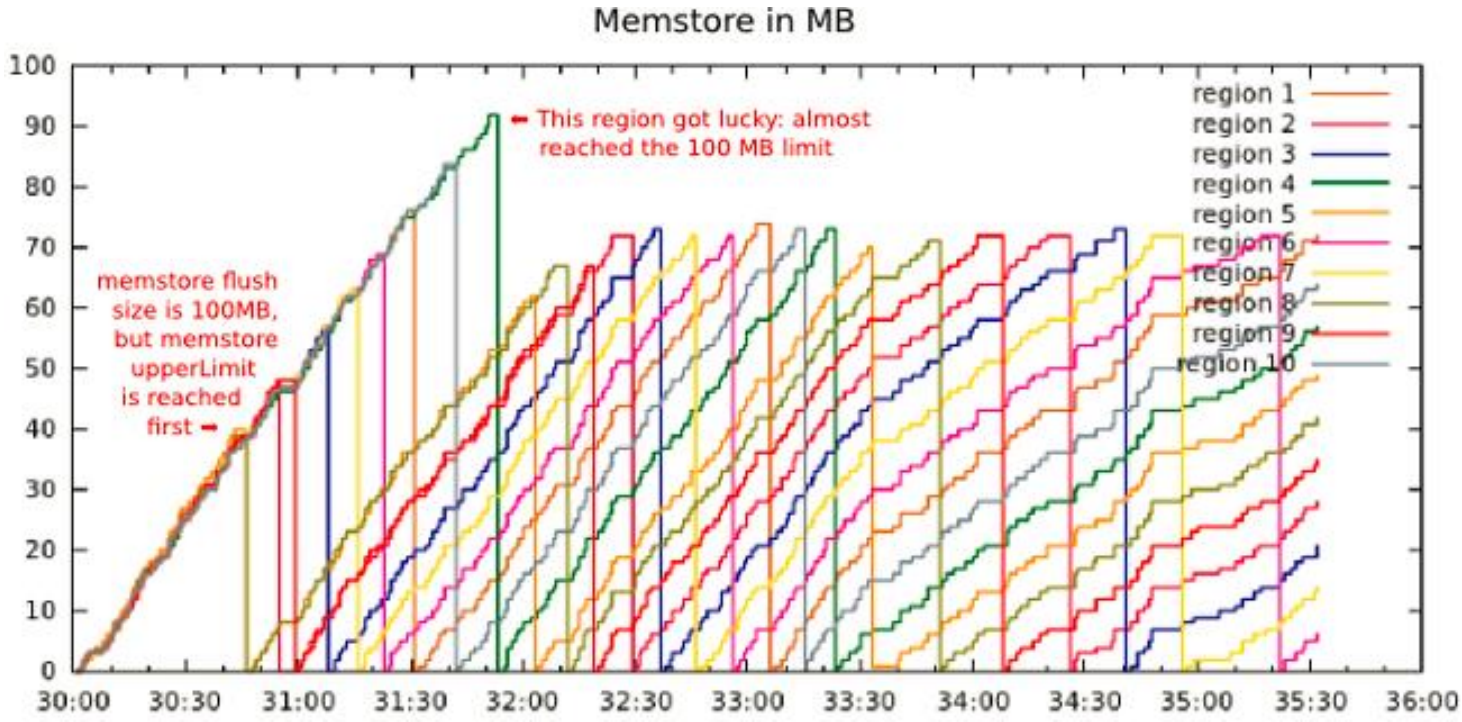
Similarly, there is one thread for doing the compactions and splits, so these are performed in sequence too. Still, it is a period where resources are used which would otherwise have been usable to serve client requests. If you would run a similar scenario a cluster, the different nodes could still run there compactions in parallel. In such a scenario, were all regions are filled very evenly and decide to do compactions at about the same time, you can see some negative spikes when observing the hbaseRequestCount metric. In the chart here you can notice that memstore grows a bit less steep during the compactions.





场景五： Memstore upperLimit

All the above illustrations were quite synthetic. On a real system, there will often be much more regions, and more than one table, and there will often not be enough memory to let memory stores grow to their configured flush size.
HBase uses by default 40% of the heap (see property hbase.regionserver.global.memstore.upperLimit) for all memstores of all regions of all column families of all tables. If this limit is reached, it starts flushing some memstores until the memory used by memstores is below at least 35% of heap (lowerLimit property).
Let's simulate this scenario by again using 10 regions, but now with an upper limit of 100 MB. The regionserver in this test has only 1GB of memory, so 400MB available for the memstores. Given that we have 10 memstores (1 table, 1 column family, 10 regions), and that we will fill them evenly, we expect the first flushes to happen when the memstores reach 400/10=40MB (instead of going on to their 100MB flush size).
And, hurary huray, the theory is confirmed by the plot below.



参考文献

An In-Depth Look at the HBase Architecture

https://www.mapr.com/blog/in-depth-look-hbase-architecture

http://blog.jobbole.com/91913/