第三周领教直播 · HBase

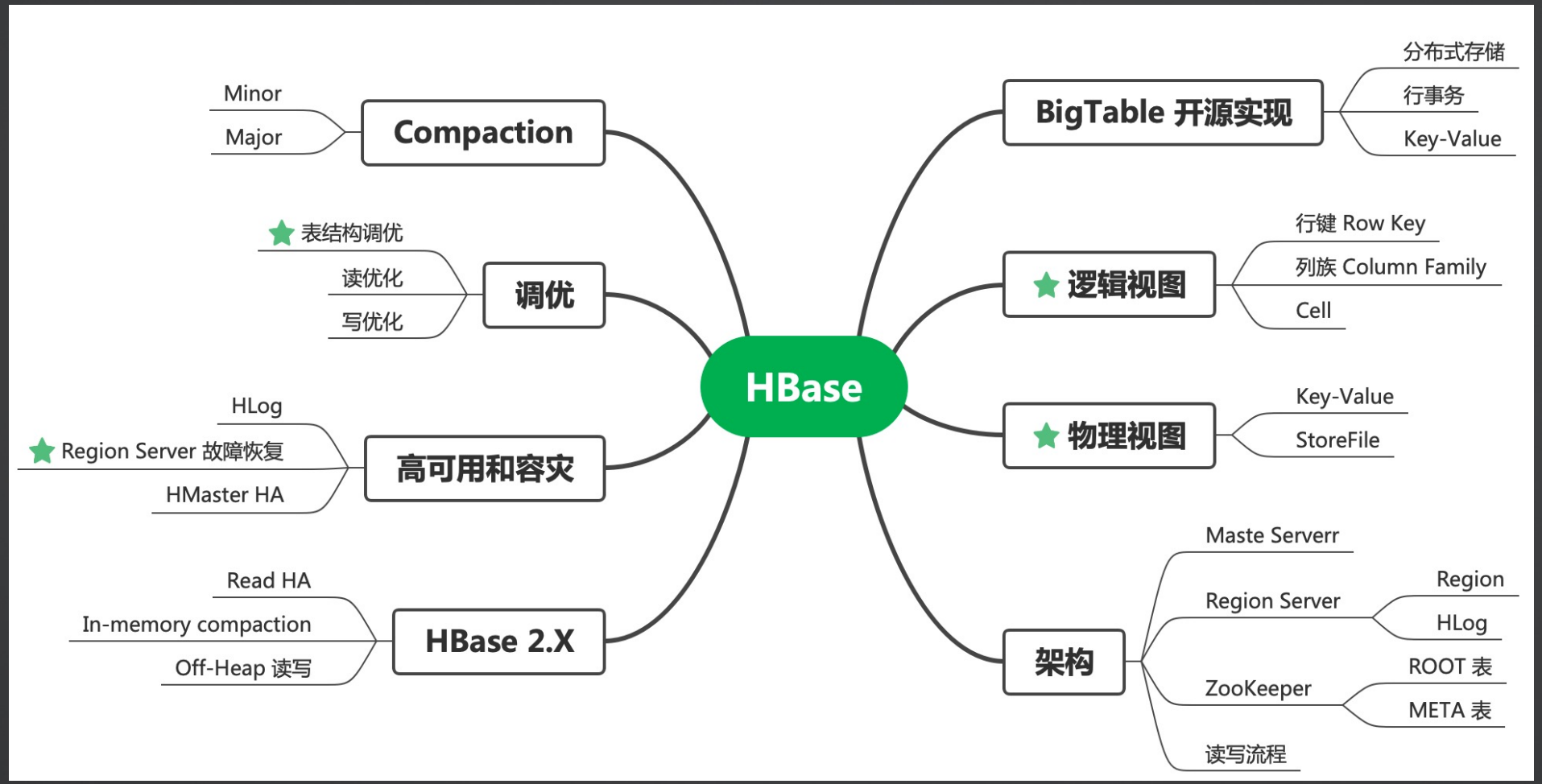
张语

极客时间训练营

目录

- 重点内容回顾
- 作业讲解
- Region Split 原理
- RowKey 设计
- QA

重点内容回顾


HBase
Compaction
Minor
Major
调优
表结构调优
读优化
写优化
高可用和容灾
HLog
Region Server 故障恢复
HMaster HA
HBase 2.X
Read HA
In-memory compaction
Off-Heap 读写
BigTable 开源实现
分布式存储
行事务
Key-Value
逻辑视图
行键 Row Key
列族 Column Family
Cell
物理视图
Key-Value
StoreFile
架构
Maste Serverr
Region Server
Region
HLog
ZooKeeper
ROOT 表
META 表
读写流程

重点内容回顾

- HBase 索引排序键由 行 + 列 + 时间戳 组成，HBase 表可以被看做一个“**稀疏的、分布式的、持久的、多维度有序 Map**”。
- HBase 只支持单行事务，即对同一行的 Put 操作保证 ACID。
- 扩展内容：
  - Spanner 可以做到多数据表事务一致性管理。

HBase

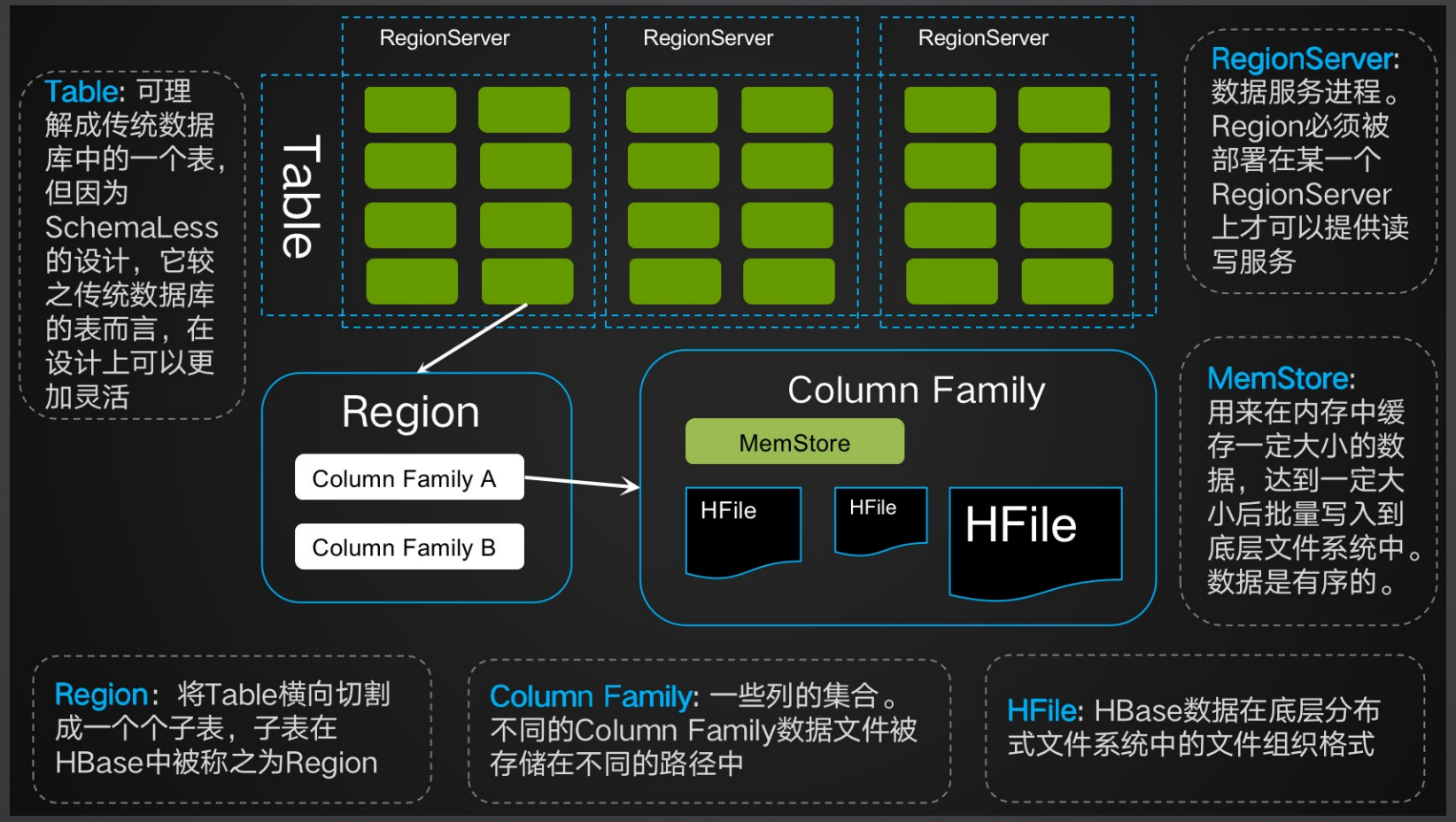


Table: 可理解成传统数据库中的一个表，但因为SchemaLess的设计，它较之传统数据库的表而言，在设计上可以更加灵活

RegionServer: 数据服务进程。Region必须被部署在某一个RegionServer上才可以提供读写服务

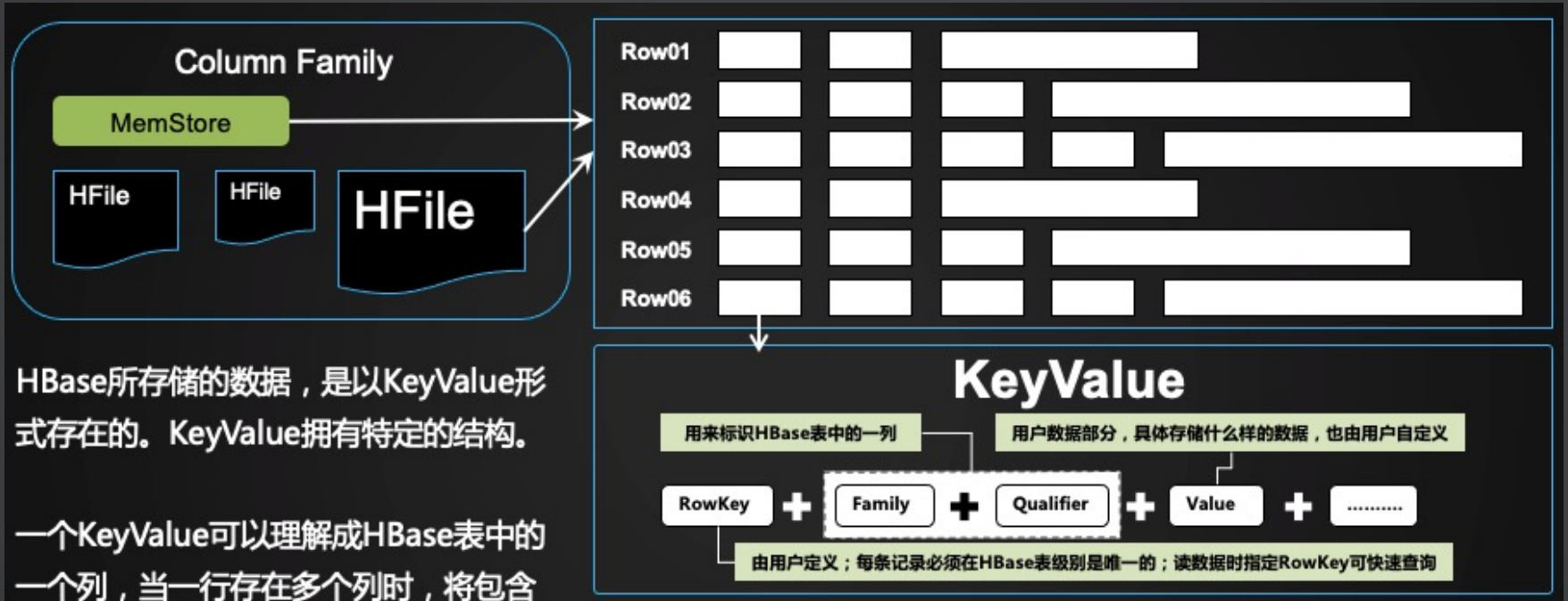
MemStore: 用来在内存中缓存一定大小的数据，达到一定大小后批量写入到底层文件系统中。数据是有序的。

Region：将Table横向切割成一个个子表，子表在HBase中被称之为Region

Column Family: 一些列的集合。不同的Column Family数据文件被存储在不同的路径中

HFile: HBase数据在底层分布式文件系统中的文件组织格式

理解 KeyValue

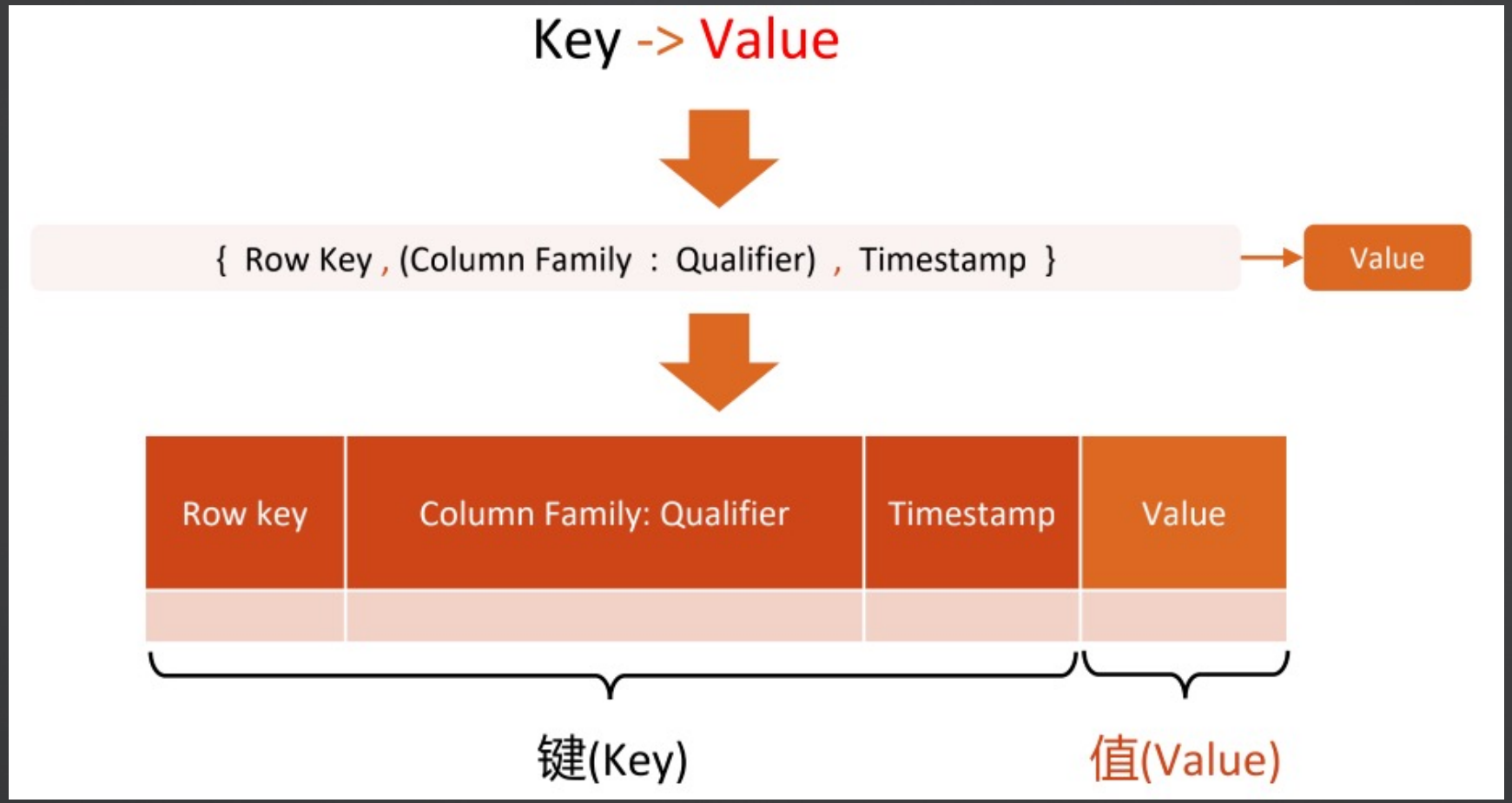


HBase所存储的数据，是以KeyValue形式存在的。KeyValue拥有特定的结构。

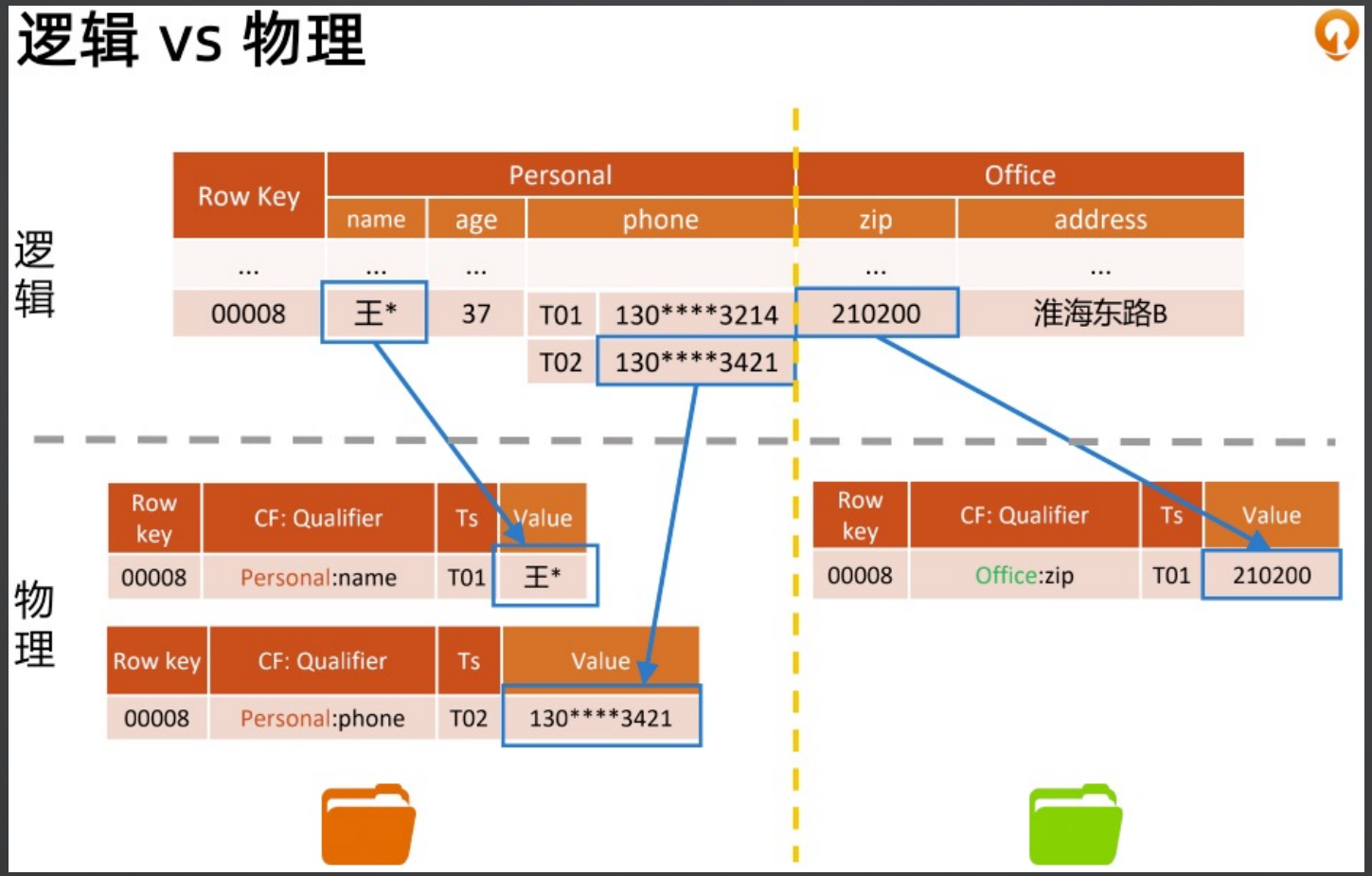
一个KeyValue可以理解成HBase表中的一个列，当一行存在多个列时，将包含多个KeyValue。同一行的KeyValue有可能存在于不同的文件中，但在读取的时候，将会按需合并在一起返回给客户端。

用户写数据时，需要定义用户数据的RowKey，指定每一列所存放的Column Family，并且为其定义相应的Qualifier(列名)，Value部分存放用户数据数据。HBase中每一行可拥有不同的KeyValues，这就是HBase Schema-less的特点。

重点内容回顾-KeyValue



重点内容回顾-视图


逻辑 vs 物理
逻辑
物理
Row Key
Personal
Office
name
age
phone
zip
address
...
00008
王*
37
T01
130****3214
210200
淮海东路B
T02
130****3421
Row key
CF: Qualifier
Ts
Value
00008
Personal:name
T01
王*
00008
Personal:phone
T02
130****3421
00008
Office:zip
T01
210200

作业讲解

使用 Java API 操作 HBase

建表，实现插入数据，删除数据，查询等功能。建立一个如下所示的表：

- 表名：$your_name:student
- 空白处自行填写，姓名学号一律填写真实姓名和学号

name	info		score	
	student_id	class	understanding	programming
Tom	20210000000001	1	75	82
Jerry	20210000000002	1	85	67
Jack	20210000000003	2	80	80
Rose	20210000000004	2	60	61
$your_name	$your_student_id			

作业讲解

```
private static final Logger logger = LoggerFactory.getLogger(HBaseOperation.class);
public static final String OP_ROW_KEY = "ZhangYu";
private static Connection connection = null;
private static Admin admin = null;

static {
    try {
        Configuration configuration = HBaseConfiguration.create();
        configuration.set("hbase.zookeeper.quorum", "emr-worker-2,emr-worker-1,emr-header-1");
        configuration.set("hbase.zookeeper.property.clientPort", "2181");
        connection = ConnectionFactory.createConnection(configuration);
        admin = connection.getAdmin();
    } catch (IOException e) {
        logger.error("init failed", e);
    }
}
```

作业讲解

```java
public static boolean createTable(String tableName, String... columnFamilies) {
    if (StringUtils.isEmpty(tableName) || columnFamilies.length < 1) {
        throw new IllegalArgumentException("tableName or columnFamilies is null");
    }

    TableDescriptorBuilder tDescBuilder =
            TableDescriptorBuilder.newBuilder(TableName.valueOf(tableName));
    for (String columnFamily : columnFamilies) {
        ColumnFamilyDescriptor descriptor =
                ColumnFamilyDescriptorBuilder.newBuilder(Bytes.toBytes(columnFamily)).build();
        tDescBuilder.setColumnFamily(descriptor);
    }

    try {
        admin.createTable(tDescBuilder.build());
        logger.info("createTable success, tableName: {}", tableName);
        return true;
    } catch (IOException e) {
        logger.error("createTable failed, tableName: {}", tableName, e);
    }

    return false;
}
```

作业讲解

```
public static void deleteTable(String tableName) throws IOException {
    admin.disableTable(TableName.valueOf(tableName));
    admin.deleteTable(TableName.valueOf(tableName));
    logger.info("deleteTable success, tableName: {}", tableName);
}

public static void putData(String tableName, String rowKey, String colFamily,
                           String colKey, String colValue) throws IOException {
    Table table = connection.getTable(TableName.valueOf(tableName));
    Put put = new Put(Bytes.toBytes(rowKey));
    put.addColumn(Bytes.toBytes(colFamily), Bytes.toBytes(colKey), Bytes.toBytes(colValue));
    table.put(put);
    table.close();
}
```

作业讲解

```
public static void getData(String tableName, String rowKey, String colFamily, String colKey) throws IOException {
    Table table = connection.getTable(TableName.valueOf(tableName));
    Get get = new Get(Bytes.toBytes(rowKey));
    if (StringUtils.isEmpty(colKey)) {
        get.addFamily(Bytes.toBytes(colFamily));
    } else {
        get.addColumn(Bytes.toBytes(colFamily), Bytes.toBytes(colKey));
    }

    Result result = table.get(get);
    for (Cell cell : result.rawCells()) {
        String family = Bytes.toString(CellUtil.cloneFamily(cell));
        String qualifier = Bytes.toString(CellUtil.cloneQualifier(cell));
        String value = Bytes.toString(CellUtil.cloneValue(cell));
        logger.info("Family:{}, Qualifier:{}, Value:{}", family, qualifier, value);
    }

    table.close();
}
```

作业讲解

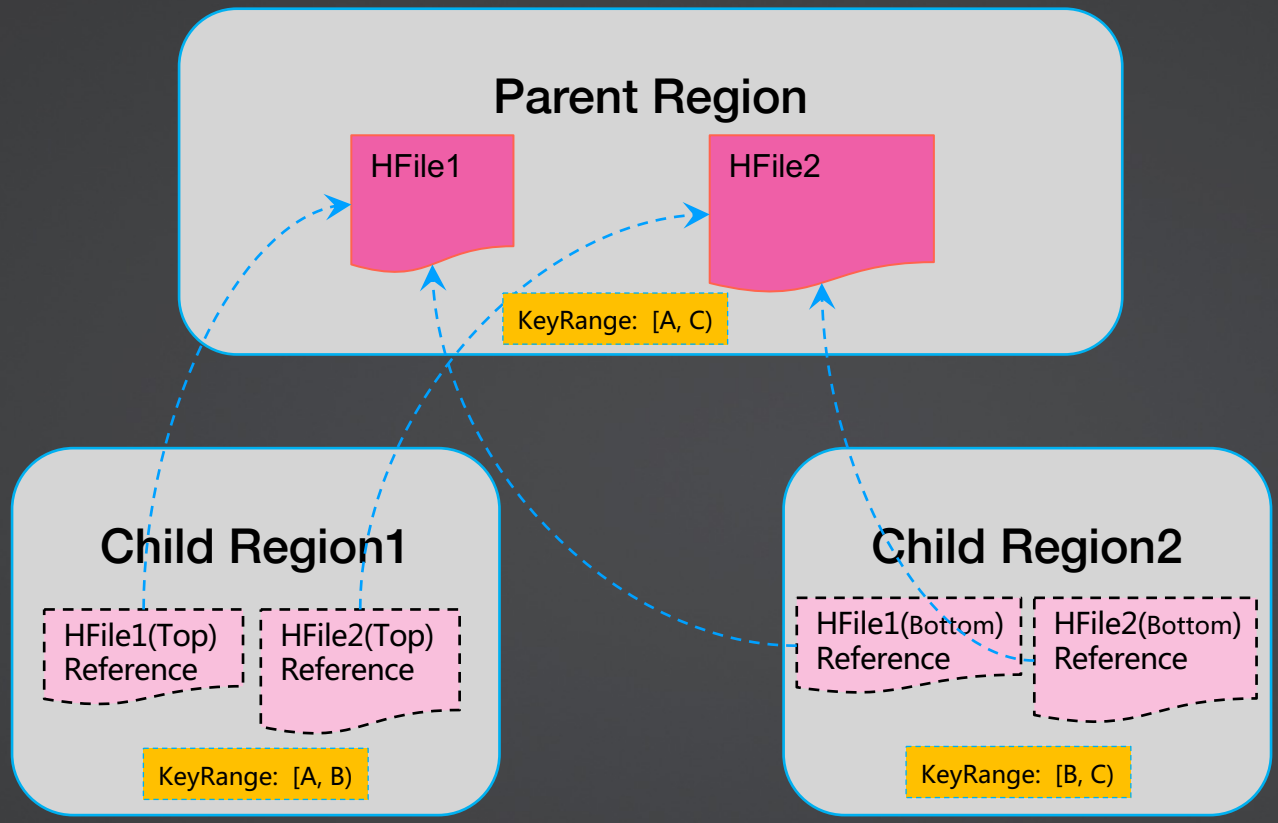
```
public static void scanTable(String tableName) throws IOException {
    Table table = connection.getTable(TableName.valueOf(tableName));
    Scan scan = new Scan();
    ResultScanner resultScanner = table.getScanner(scan);
    for (Result result : resultScanner) {
        for (Cell cell : result.rawCells()) {
            String row = Bytes.toString(CellUtil.cloneRow(cell));
            String family = Bytes.toString(CellUtil.cloneFamily(cell));
            String qualifier = Bytes.toString(CellUtil.cloneQualifier(cell));
            String value = Bytes.toString(CellUtil.cloneValue(cell));
            logger.info("Row:{}, Family:{}, Qualifier:{}, Value:{}", row, family, qualifier, value);
        }
    }
}

public static void deleteData(String tableName, String rowKey, String colFamily, String colKey) throws IOException {
    Table table = connection.getTable(TableName.valueOf(tableName));
    Delete delete = new Delete(Bytes.toBytes(rowKey));
    delete.addColumn(Bytes.toBytes(colFamily), Bytes.toBytes(colKey));

    table.delete(delete);
}
```

作业讲解

java -jar target/bigdata-tutorial-1.0-SNAPSHOT-jar-with-dependencies.jar com.bigdata.tutorial.hbase.HBaseOperation

Region Split

- 数据库集群负载均衡的实现依赖于数据库的数据分片设计。
- 负载均衡功能是数据分片在集群中均衡的实现。
- HBase 中数据分片的概念是 Region。
- HBase Split 根据触发条件和分裂策略将一个 region 进行分裂成两个子 region 并对父 region 进行清除处理的过程。



Region Split 过程并不会真正将父 Region 中的 HFile 数据搬到子 Region 目录中。

Split 过程仅仅是在子 Region 中创建了到父 Region 的 HFile 的引用文件，子 Region1 中的引用文件指向原 HFile 的上部，而子 Region2 的引用文件指向原 HFile2 的下部。

Region Split

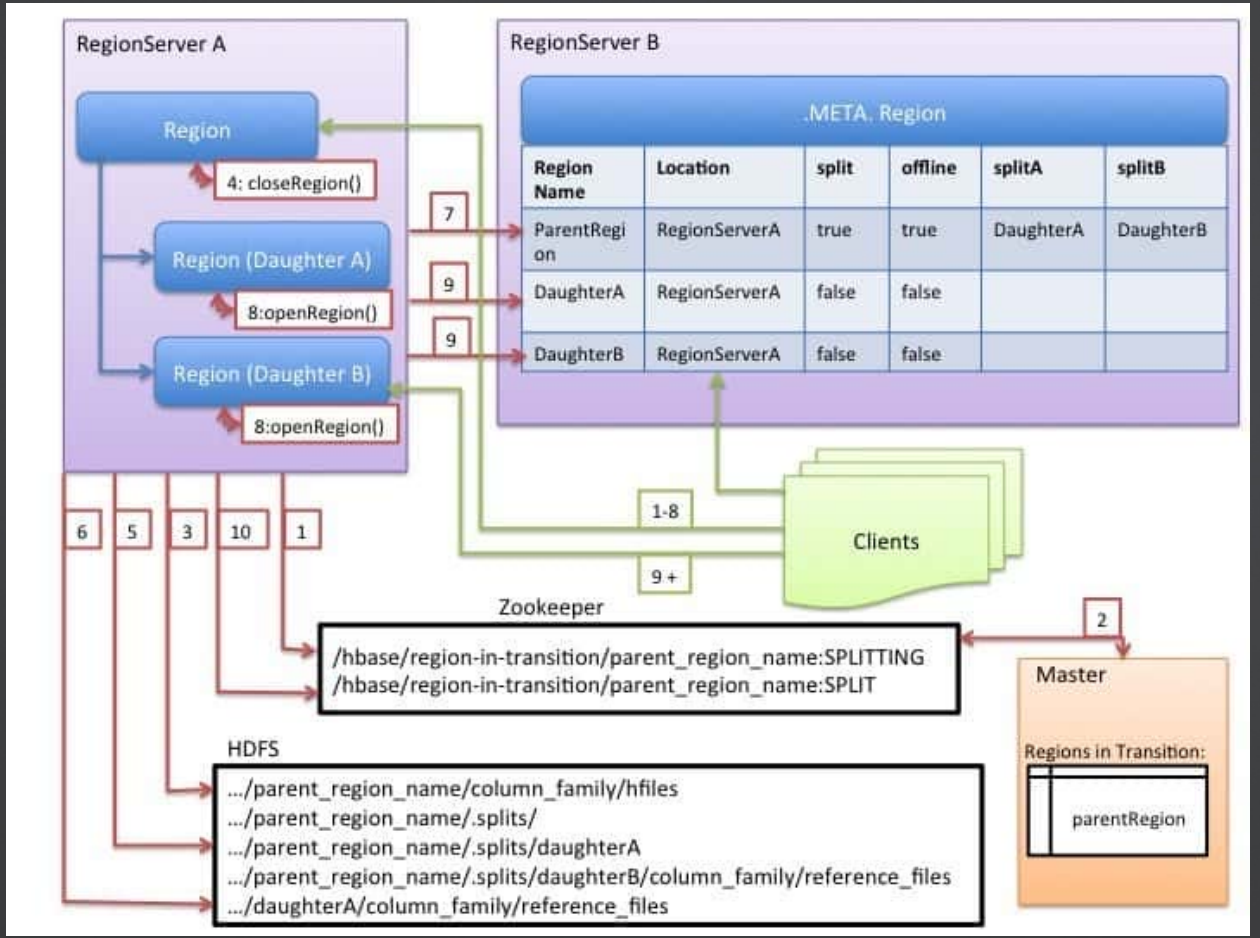
- Split 是 HBase 根据一定的触发条件和一定的分裂策略将 HBase 的一个 region 进行分裂成两个子 region 并对父 region 进行清除处理的过程。
- HBase 将整个分裂过程包装成了一个事务，目的是保证分裂事务的原子性。
- 整个分裂事务过程分为三个阶段 prepare、execute 和 rollback。

Region Split-Prepare

```java
/**
 * Information about a region. A region is a range of keys in the whole keyspace
 * of a table, an identifier (a timestamp) for differentiating between subset
 * ranges (after region split) and a replicaId for differentiating the instance
 * for the same range and some status information about the region.
 *
 * The region has a unique name which consists of the following fields:
 * <ul>
 * <li> tableName   : The name of the table </li>
 * <li> startKey    : The startKey for the region. </li>
 * <li> regionId    : A timestamp when the region is created. </li>
 * <li> replicaId   : An id starting from 0 to differentiate replicas of the
 * same region range but hosted in separated servers. The same region range can
 * be hosted in multiple locations.</li>
 * <li> encodedName : An MD5 encoded string for the region name.</li>
 * </ul>
 *
 * <br> Other than the fields in the region name, region info contains:
 * <ul>
 * <li> endKey      : the endKey for the region (exclusive) </li>
 * <li> split       : Whether the region is split </li>
 * <li> offline     : Whether the region is offline </li>
 * </ul>
 *
 */
@InterfaceAudience.Public
public interface RegionInfo extends Comparable<RegionInfo> {
```

- 在内存中初始化两个子 Region，具体生成两个 HRegionInfo 对象，包含 tableName、regionName、startkey、endkey 等。

Region Split-Execute



Region Split-Execute

1. RegionServer 将 ZooKeeper 节点/region-in-transition 中该 Region 的状态置为 SPLITING。

2. Master 通过 watch 节点 /region-in-transition 检测到 Region状态改变，并修改内存中 Region 的状态。

3. 在父存储目录下新建 .split 文件夹，保存 daughter region 信息。

4. 关闭父 Region。关闭数据写入并触发 flush 操作，将写入 Region 的数据全部持久化到磁盘。

Region Split-Execute

5. 在.split 文件夹下新建两个子文件夹（ daughter A/B），并生成 reference 文件，分别指向父 Region 中对应文件。

6. 父 Region 分裂为两个子 Region 后，将 Daughter Region A/B 移动到 HBase 根目录下，形成两个新的 Region。

7. 父 Region 通知修改 hbase:meta 表后下线，不再提供服务。

8. 开启 daughter A、daughter B 两个子 Region。通知修改 hbase:meta 表，正式对外提供服务。

Region Split Policy

```
/**
 * A split policy determines when a Region should be split.
 *
 * @see SteppingSplitPolicy Default split policy since 2.0.0
 * @see IncreasingToUpperBoundRegionSplitPolicy Default split policy since
 *      0.94.0
 * @see ConstantSizeRegionSplitPolicy Default split policy before 0.94.0
 */
@InterfaceAudience.LimitedPrivate(HBaseInterfaceAudience.CONFIG)
public abstract class RegionSplitPolicy extends Configured {
```

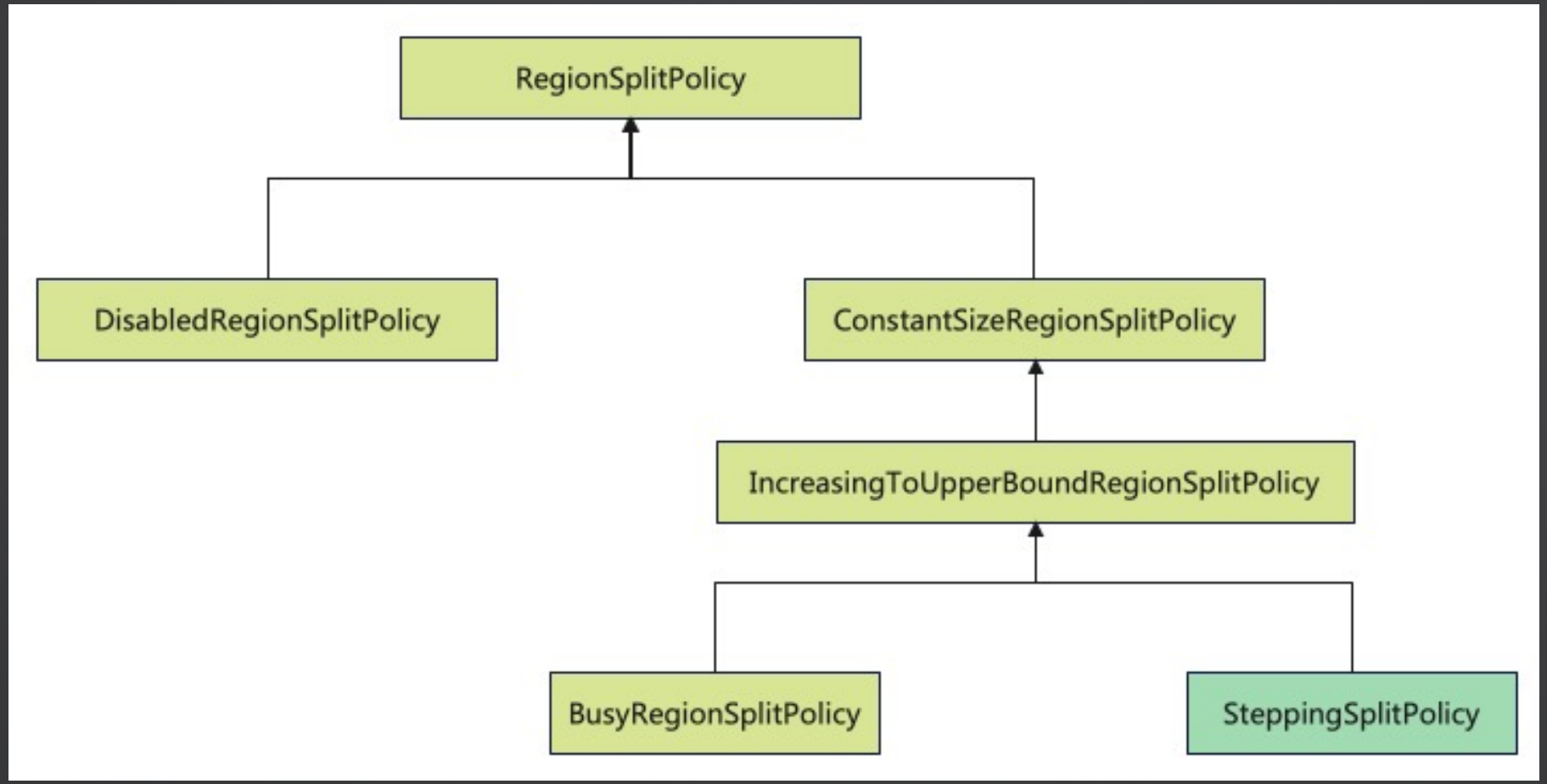
Choose Subclass of **RegionSplitPolicy** (8 classes found)

- BusyRegionSplitPolicy (org.apache.hadoop.hbase.regionserver)
- ConstantSizeRegionSplitPolicy (org.apache.hadoop.hbase.regionserver)
- CustomSplitPolicy in TestSplitTransactionOnCluster (org.apache.hadoop.hbase.re
- ~~DelimitedKeyPrefixRegionSplitPolicy~~ (org.apache.hadoop.hbase.regionserver)
- DisabledRegionSplitPolicy (org.apache.hadoop.hbase.regionserver)
- IncreasingToUpperBoundRegionSplitPolicy (org.apache.hadoop.hbase.regionserver)
- ~~KeyPrefixRegionSplitPolicy~~ (org.apache.hadoop.hbase.regionserver)
- SteppingSplitPolicy (org.apache.hadoop.hbase.regionserver)

Region Split Policy

- Region 分裂过程因为没有涉及数据的移动，所以分裂成本本身并不是很高，可以很快完成。
- 分裂后子 Region 的文件实际没有任何用户数据，文件中存储的仅是一些元数据信息——分裂点 rowkey 等。

Region Split Policy



Region Split Policy

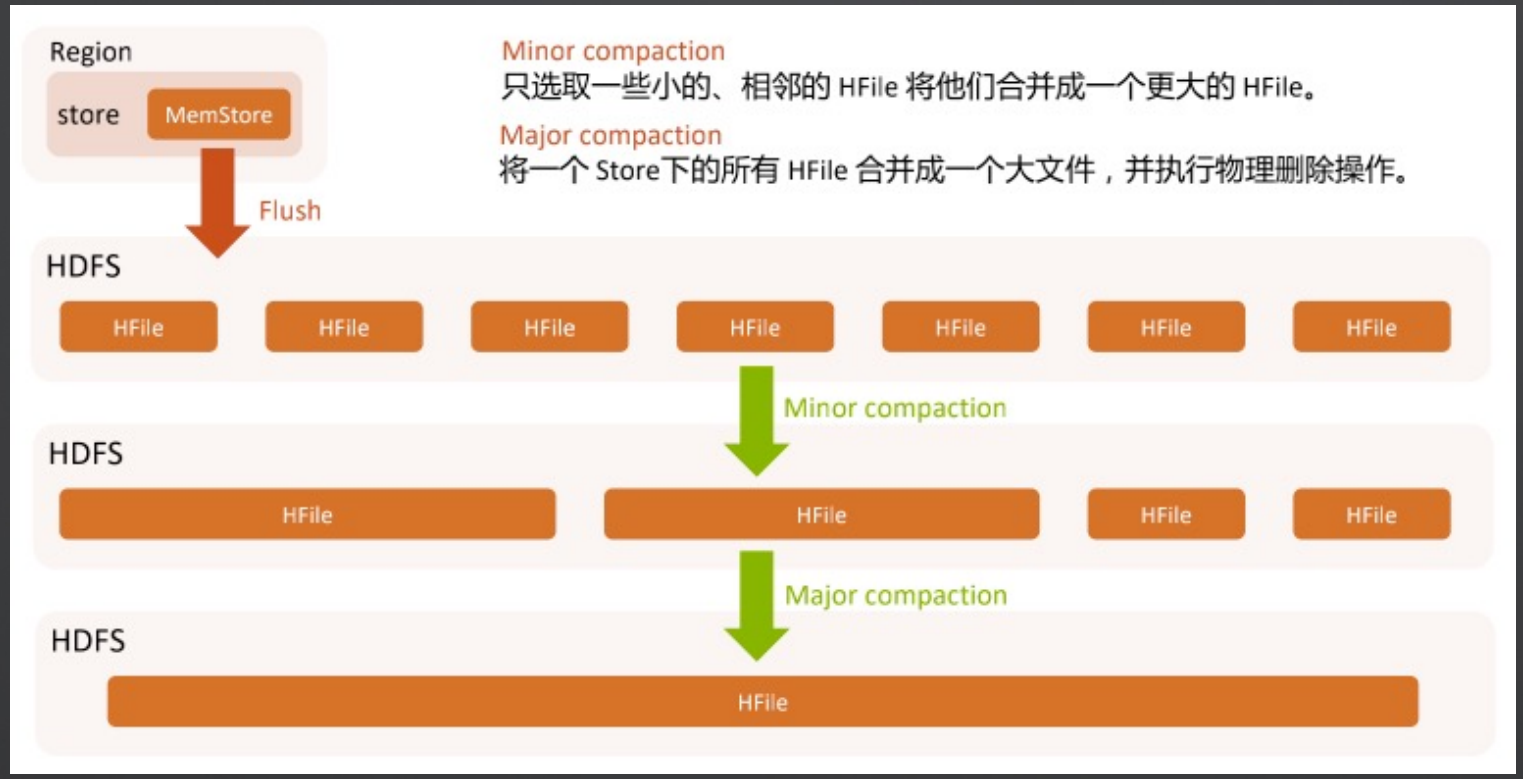
策略	原理	描述
ConstantSizeRegionSplitPolicy （0.94版本前默认策略）	当region中最大store的大小大于设置阈值之后才会触发切分（hbase.hregion.max.filesize）	对于大表和小表没有明显区分，阈值较大时小表可能不触发分裂；阈值较小时，大表会产生大量Region，影响集群得性能。
IncreasingToUpperBoundRegionSplitPolicy （0.94版本后默认策略）	阈值在一定条件下不断调整，调整规则与 Region 所属表在当前 RegionServer上的 Region 个数有关:(#regions)3 * flush size * 2，最大值为MaxRegionFileSize值。	在大集群条件下对于大表表现很优秀，但很多小表会在大集群中产生大量小region，分散在整个集群中。在发生region迁移时也可能会触发region分裂。
SteppingSplitPolicy （2.0版本后默认策略）	阈值依旧与 Region 所属表在当前 RegionServer上的 Region 个数有关：如果只有一个region，第一次拆分为256MB，之后均为MaxRegionFileSize值。	这种切分策略对于大集群中的大表、小表较之前的策略更加友好，将小表的Region控制在一个合理的范围，对大表的拆分也不影响。

RowKey 作用-读写流程

- 读写数据时通过 RowKey 路由到对应的 Region。
- MemStore 中的数据按 RowKey 字典序排序。
- HFile 中的数据按 RowKey 字典序排序。

RowKey 作用-Compaction

- 如果所有 Region 都是写活跃的，Major Compaction 的数据总量随着时间的推移不断增大



RowKey 也是索引

- 聚集索引
  - 以主键创建的索引，表记录的排列顺序和索引的排列顺序一致。
  - 优点：查询效率快；缺点：在区间插入时需要对数据页重新排序。
- 非聚集索引（二级索引）
  - 索引的逻辑顺序与磁盘上行的物理存储顺序不同，非聚集索引在叶子节点存储的是主键和索引列，当使用非聚集索引查询数据时，需要拿到叶子上的主键再去表中查找数据。
- 聚集索引和非聚集索引的区别
  - 聚集索引在叶子节点存储的是表中的数据。
  - 非聚集索引在叶子节点存储的是主键和索引列。

索引设计

- 负责特点：写多读少、读多写少
- 查询场景：高频的查询场景、时延要求
- 数据特点：离散度、数据分布特点

- RowKey：唯一标识一行数据（主键）
- 最高频的查询场景
- 组合字段考虑字段顺序
  - 参考联合索引

数据热点

- RowKey 的行由行键按字典顺序排序，这样的设计优化了扫描。
- 不好的 RowKey，大量流量流向集群上的少数节点，出现热点。
- 为了防止在写操作时出现热点，设计 RowKey 时应该使得数据尽量往多个地域上写。

避免数据热点-Salting

- 在原 Rowkey 前加固定长度的随机数，保障数据在所有 Region 间的负载均衡
- 优点：提高了写吞吐量；对读操作不优化，可能需要扫所有 Region。

原 Rowkey（手机号）	新 Rowkey
13512341234	A13512341234
13512344321	B13512344321
13612124545	C13612124545

避免数据热点-翻转

- RowKey 尾部的数据呈现出良好的随机性，可以将 RowKey 翻转，或者直接将尾部的数据放到前面。
- 优点：写吞吐量高、Get 操作可预测（相同的翻转规则）。
- 缺点：打乱了原 RowKey 的顺序，无法 Scan。

原 Rowkey（手机号）	新 Rowkey
13512341234	12341351234
13512344321	43211351234
13612124545	45451361212

避免数据热点- Hashing

- 基于 RowKey 的完整或部分数据进行 Hash，而后将 Hashing 后的值完整替换原 RowKey 或部分替换 RowKey 的前缀部分。
- 优点：提高了写吞吐量；读操作能够预测（使用同一 hash 函数）。
- 缺点：打乱了原 RowKey 的顺序，无法 Scan。

原 Rowkey（手机号）	新 Rowkey
13512341234	ABCD
13512344321	BCDE
13612124545	CDEF

面试考点分析-HBase

- RowKey 的设计原则：见上面的分析
- HBase 的适用场景
  - 分布式、 NoSQL 数据库，支持对大数据进行随机、实时读写访问
  - 数据存储在 HDFS 上
- HBase 缺点
  - 复杂查询条件性能差，原生不支持二级索引
  - 只支持单行事务

面试考点分析-HBase

- HBase 替代/类似方案
  - Cassandra：NoSQL分布式数据库。https://cassandra.apache.org/
  - TiDB：一个 Spanner 开源实现，兼容 MySQL 5.7 协议

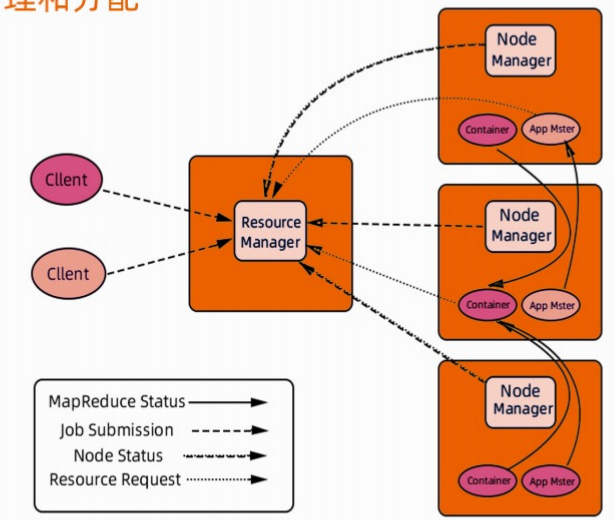
  https://docs.pingcap.com/zh/tidb/stable/overview

面试考点分析-Yarn

- YARN 设计思路和架构

YARN 的基本组成

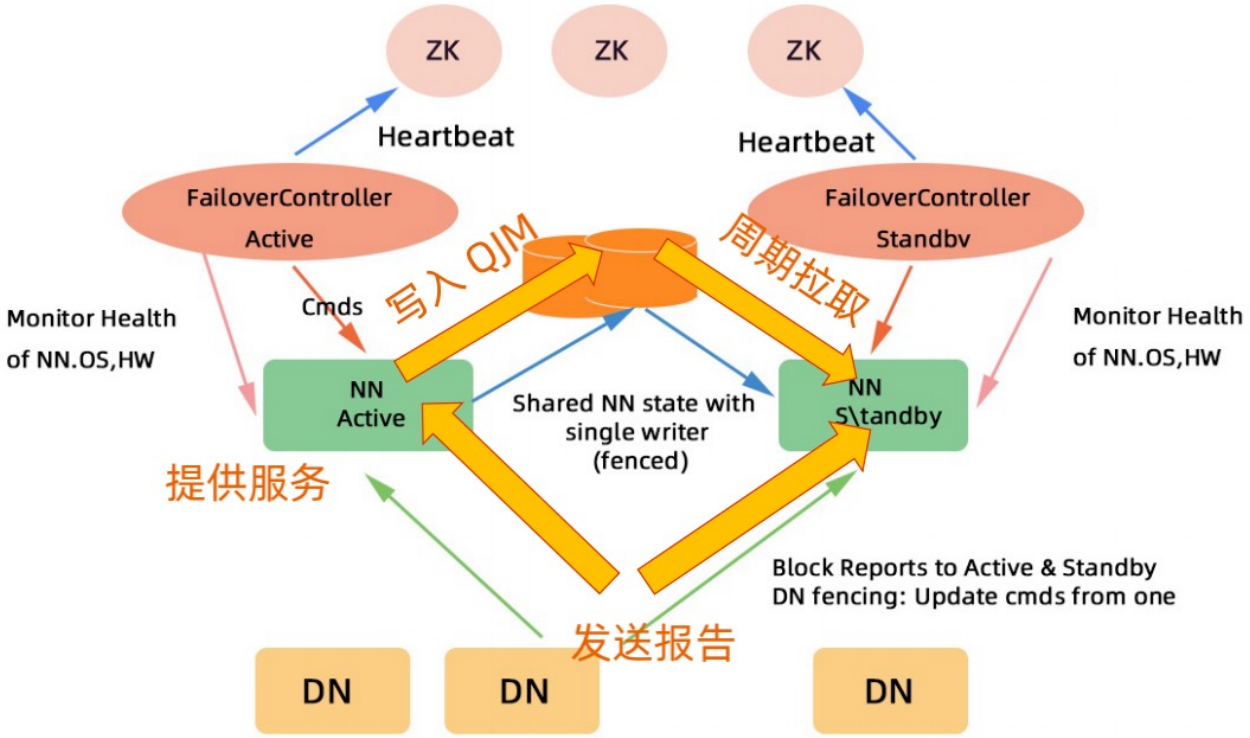
- ResourceManager：全局的资源管理器，负责整个系统的资源管理和分配
  - 处理客户端请求
  - 启动/监控 ApplicationMaster
  - 监控 NodeManager
  - 资源分配和调度
- NodeManager：驻留在一个 YARN 集群中的每个节点上的代理
  - 单个节点的资源管理
  - 处理来自 ResourceManger 的命令
  - 处理来自 ApplicationMaster 的命令
- ApplicationMaster：应用程序管理器，负责系统中所有应用程序的管理工作
  - 数据切分
  - 为应用程序申请资源，并进行分配
  - 任务监控和容错



面试考点分析-HDFS

如何保持主和备 NameNode 的状态同步？

- Active NameNode 启动后提供服务，并把 Editlog 写到本地和 QJM* 中。
- Standby NameNode 周期性的从 QJM 中拉取 Editlog，与 active 的状态保持同步。
- DataNode 同时向两个 NameNode 发送 BlockReport。



QA