## **RowKey的作用**

### **RowKey在查询中的作用**

HBase中RowKey可以唯一标识一行记录，在HBase中检索数据有以下三种方式：

1. 通过 **get** 方式，指定 **RowKey** 获取唯一一条记录
2. 通过 **scan** 方式，设置 **startRow** 和 **stopRow** 参数进行范围匹配
3. **全表扫描**，即直接扫描整张表中所有行记录

当大量请求访问HBase集群的一个或少数几个节点，造成少数RegionServer的读写请求过多、负载过大，而其他RegionServer负载却很小，这样就造成**热点现象**。大量访问会使热点Region所在的主机负载过大，引起性能下降，甚至导致Region不可用。所以我们在向HBase中插入数据的时候，应尽量均衡地把记录分散到不同的Region里去，平衡每个Region的压力。

下面根据一个例子分别介绍下根据RowKey进行查询的时候支持的情况。

如果我们RowKey设计为uid+phone+name，那么这种设计可以很好的支持一下的场景:

uid=873969725 AND phone=18900000000 AND name=zhangsan

uid= 873969725 AND phone=18900000000

uid= 873969725 AND phone=189?

uid= 873969725

难以支持的场景：

phone=18900000000 AND name = zhangsan

phone=18900000000

name=zhangsan

从上面的例子中可以看出，在进行查询的时候，根据RowKey从前向后匹配，所以我们在设计RowKey的时候选择好字段之后，还应该结合我们的实际的高频的查询场景来组合选择的字段，越高频的查询字段排列越靠左。

**索引设计**

数据库查询可简单分解为两个步骤：1）键的查找；2) 数据的查找

因这两种数据组织方式的不同，在RDBMS领域有两种常见的数据组织表结构：

**索引组织表：**键与数据存放在一起，查找到键所在的位置则意味着查找到数据本身。

**堆表：**键的存储与数据的存储是分离的。查找到键的位置，只能获取到数据的物理地址，还需要基于该地址去获取数据。

HBase数据表其实是一种**索引组织表结构**：查找到**RowKey**所在的位置则意味着找到数据本身。因此，**RowKey本身就是一种索引**。

### **RowKey在Region中的作用**

在 HBase 中，Region 相当于一个数据的分片，每个 Region 都有StartRowKey和StopRowKey，这是表示 Region 存储的 RowKey 的范围，HBase 表的数据时按照 RowKey 来分散到不同的 Region，要想将数据记录均衡的分散到不同的Region中去，因此需要 RowKey 满足这种散列的特点。此外，在数据读写过程中也是与RowKey 密切相关，RowKey在读写过程中的作用：

1. 读写数据时通过 RowKey 找到对应的 Region；
2. MemStore 中的数据是按照 RowKey 的字典序排序；
3. HFile 中的数据是按照 RowKey 的字典序排序。

## **RowKey的设计**

在HBase中RowKey在数据检索和数据存储方面都有重要的作用，一个好的RowKey设计会影响到数据在HBase中的分布，还会影响我们查询效率，所以一个好的RowKey的设计方案是多么重要。首先我们先来了解下RowKey的设计原则。

### **RowKey设计原则**

**长度原则**

RowKey是一个二进制码流（所有存储在HBase表中的数据都是二进制的字节），可以是任意字符串，最大长度为64kb，实际应用中一般为10-100byte，以byte[]形式保存，一般设计成定长。建议越短越好，不要超过16个字节，原因如下：

1数据的持久化文件HFile中时按照Key-Value存储的，如果RowKey过长，例如超过100byte，那么1000w行的记录，仅RowKey就需占用近1GB的空间。这样会极大影响HFile的存储效率。

2.MemStore会缓存部分数据到内存中，若RowKey字段过长，内存的有效利用率就会降低，就不能缓存更多的数据，从而降低检索效率。

3.目前操作系统都是64位系统，内存8字节对齐，控制在16字节，8字节的整数倍利用了操作系统的最佳特性。

**唯一原则**

必须在设计上保证RowKey的唯一性。由于在HBase中数据存储是Key-Value形式，若向HBase中同一张表插入相同RowKey的数据，则原先存在的数据会被新的数据覆盖。

**排序原则**

HBase的RowKey（同一列的元素）是按照ASCII有序排序的(升序)，因此我们在设计RowKey的时候要充分利用这点。

**散列原则**

设计的RowKey应均匀的分布在各个HBase节点上。

### **RowKey字段选择**

RowKey字段的选择，遵循的最基本原则是唯一性，RowKey必须能够唯一的识别一行数据。无论应用的负载特点是什么样，RowKey字段都应该参考最高频的查询场景。数据库通常都是以如何高效的读取和消费数据为目的，而不是数据存储本身。然后，结合具体的负载特点，再对选取的RowKey字段值进行改造，组合字段场景下需要重点考虑字段的顺序。

### **避免数据热点的方法**

在对HBase的读写过程中，如何避免热点现象呢？主要有以下几种方法：

**Reversing**

如果经初步设计出的RowKey在数据分布上不均匀，但RowKey尾部的数据却呈现出了良好的随机性，此时，可以考虑将RowKey的信息翻转，或者直接将尾部的bytes提前到RowKey的开头。Reversing可以有效的使RowKey随机分布，但是牺牲了RowKey的有序性。

缺点：

利于Get操作，但不利于Scan操作，因为数据在原RowKey上的自然顺序已经被打乱。

**Salting**

Salting（加盐）的原理是在原RowKey的前面添加固定长度的随机数，也就是给RowKey分配一个随机前缀使它和之间的RowKey的开头不同。随机数能保障数据在所有Regions间的负载均衡。

缺点：

因为添加的是随机数，基于原RowKey查询时无法知道随机数是什么，那样在查询的时候就需要去各个可能的Regions中查找，Salting对于读取是利空的。并且加盐这种方式增加了读写时的吞吐量。

**Hashing**

基于 RowKey 的完整或部分数据进行 Hash，而后将Hashing后的值完整替换或部分替换原RowKey的前缀部分。这里说的 hash 包含 MD5、sha1、sha256 或 sha512 等算法。

缺点：

与 Reversing 类似，Hashing 也不利于 Scan，因为打乱了原RowKey的自然顺序。

## **RowKey设计案例剖析**

**1. 查询某用户在某应用中的操作记录**

reverse(userid) + appid + timestamp

**2. 查询某用户在某应用中的操作记录（优先展现最近的数据）**

reverse(userid) + appid + (Long.Max\_Value - timestamp)

**3. 查询某用户在某段时间内所有应用的操作记录**

reverse(userid) + timestamp + appid

**4. 查询某用户的基本信息**

reverse(userid)

**5. 查询某eventid记录信息**

salt + eventid + timestamp

如果 userid是按数字递增的，并且长度不一，可以先预估 userid 最大长度，然后将userid进行翻转，再在翻转之后的字符串后面补0（至最大长度）；如果长度固定，直接进行翻转即可（如手机号码）。

在第5个例子中，加盐的目的是为了增加查询的并发性，加入Slat的范围是0~n，可以将数据分为n个split同时做scan操作，有利于提高查询效率。

## **总结**

在HBase的使用过程，设计RowKey是一个很重要的一个环节。我们在进行RowKey设计的时候可参照如下步骤：

1. 结合业务场景特点，选择合适的字段来做为RowKey，并且按照查询频次来放置字段顺序
2. 通过设计的RowKey能尽可能的将数据打散到整个集群中，均衡负载，避免热点问题
3. 设计的RowKey应尽量简短