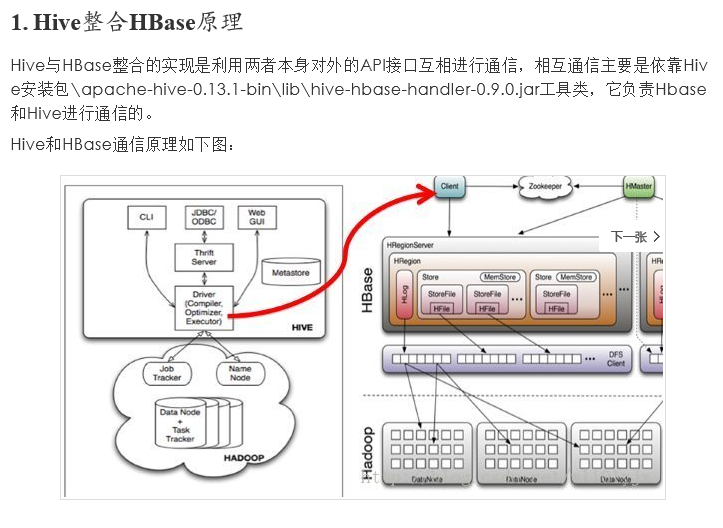
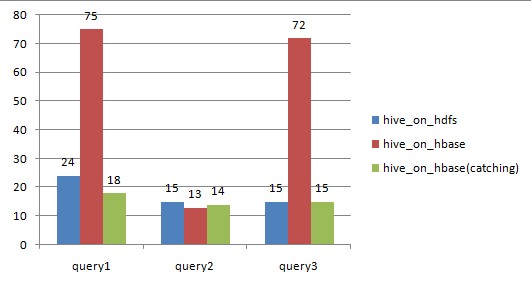
# **Hive与Hbase结合的原理**



# 一、查询性能比较：

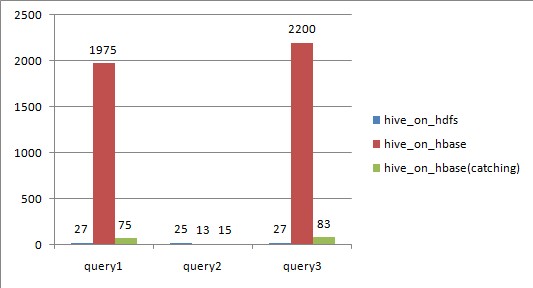
query1:  
select count(1) from on\_hdfs;  
select count(1) from on\_hbase;  
query2(根据key过滤)  
select \* from on\_hdfs  
where key = ‘13400000064\_1388056783\_460095106148962′;  
select \* from on\_hbase  
where key = ‘13400000064\_1388056783\_460095106148962′;  
query3(根据value过滤)  
select \* from on\_hdfs where value = ‘XXX';  
select \* from on\_hbase where value = ‘XXX';

on\_hdfs (20万记录，150M，TextFile on HDFS)  
on\_hbase(20万记录，160M，HFile on HDFS)

[](http://lxw1234.com/wp-content/uploads/2015/04/0417-12.jpg)

Hive over HBase

on\_hdfs (2500万记录，2.7G，TextFile on HDFS)  
on\_hbase(2500万记录，3G，HFile on HDFS)

[](http://lxw1234.com/wp-content/uploads/2015/04/0417-13.jpg)

Hive over HBase

从上图可以看出，  
对于全表扫描，hive\_on\_hbase查询时候如果不设置catching，性能远远不及hive\_on\_hdfs；  
根据rowkey过滤，hive\_on\_hbase性能上略好于hive\_on\_hdfs，特别是数据量大的时候；  
设置了caching之后，尽管比不设caching好很多，但还是略逊于hive\_on\_hdfs；

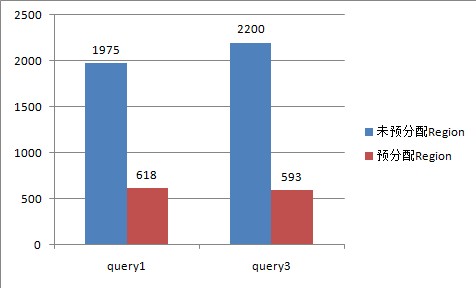
# 二、Hive over HBase原理

**Hive与Hbase和mapreduce** 利用两者本身对外的API来实现整合，主要是靠HBaseStorageHandler进行通信，利用 HBaseStorageHandler，Hive可以获取到Hive表对应的HBase表名，列簇以及列，InputFormat和 OutputFormat类，创建和删除HBase表等。Hive访问HBase中表数据，实质上是通过**MapReduce读取HBase表数据**，其实先是在MR中，使用HiveHBaseTableInputFormat完成对HBase表的切分，获取RecordReader对象来读取数据。  
对HBase表的切分原则是一个Region切分成一个Split,即表中有多少个Regions,MR中就有多少个Mapper；读取HBase表数据都是通过构建Scanner，对表进行全表扫描，如果有过滤条件，则转化为Filter。当过滤条件为rowkey时，则转化为对rowkey的过滤；  
Scanner通过RPC调用RegionServer的next()来获取数据；

# 三、性能瓶颈分析

## 1. Map Task

Hive读取HBase表，通过MR,最终使用HiveHBaseTableInputFormat来读取数据，在getSplit()方法中对 HBase表进行切分，切分原则是根据该表对应的HRegion，将每一个Region作为一个InputSplit，即，该表有多少个Region,就 有多少个Map Task；  
每个Region的大小由参数hbase.hregion.max.filesize控制，默认10G，这样会使得每个map task处理的数据文件太大，map task性能自然很差；  
为HBase表预分配Region，使得每个Region的大小在合理的范围；  
下图是给该表预分配了15个Region，并且控制key均匀分布在每个Region上之后，查询的耗时对比，其本质上是Map数增加。

[](http://lxw1234.com/wp-content/uploads/2015/04/0417-14.jpg)

Hive over HBase

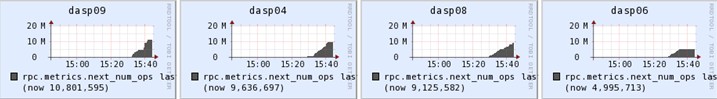
## 2. Scan RPC 调用：

* 在Scan中的每一次next()方法都会为每一行数据生成一个单独的RPC请求， query1和query3中，全表有2500万行记录，因此要2500万次RPC请求；

* 扫描器缓存（Scanner Caching）：HBase为扫描器提供了缓存的功能，可以通过参数hbase.client.scanner.caching来设置；默认是100；缓存 的原理是通过设置一个缓存的行数，当客户端通过RPC请求RegionServer获取数据时，RegionServer先将数据缓存到内存，当缓存的数据行数达到参数设置的数量时，再一起返回给客户端。这样，通过设置扫描器缓存，就可以大幅度减少客户端RPC调用RegionServer的次数；但并不 是缓存设置的越大越好，如果设置的太大，每一次RPC调用将会占用更长的时间，因为要获取更多的数据并传输到客户端，如果返回给客户端的数据超出了其堆的 大小，程序就会终止并抛出OOM异常；

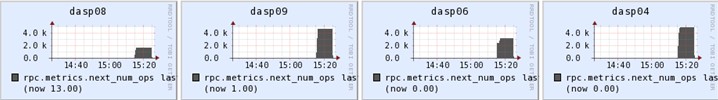
所以，需要为少量的RPC请求次数和客户端以及服务端的内存消耗找到平衡点。

rpc.metrics.next\_num\_ops  
未设置caching,每个RegionServer上通过next()方法调用RPC的次数峰值达到1000万：

[](http://lxw1234.com/wp-content/uploads/2015/04/0417-15.jpg)

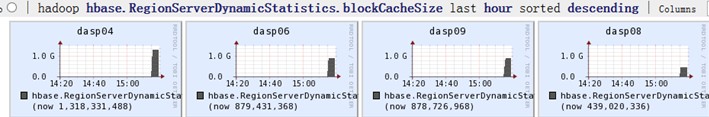
Hive over HBase

设置了caching=2000，每个RegionServer上通过next()方法调用RPC的次数峰值只有4000：

[](http://lxw1234.com/wp-content/uploads/2015/04/0417-16.jpg)

Hive over HBase

设置了caching之后，几个RegionServer上的内存消耗明显增加：

[](http://lxw1234.com/wp-content/uploads/2015/04/0417-17.jpg)

Hive over HBase

* 扫描器批量（Scanner Batch）：缓存是面向行一级的操作，而批量则是面向列一级的操作。批量可以控制每一次next()操作要取回多少列。比如，在扫描器中设置setBatch(5),则一次next()返回的Result实例会包括5列。
* RPC请求次数的计算公式如下：  
  RPC请求次数 =  
  （表行数 \* 每行的列数）/ Min(每行的列数，批量大小)  / 扫描器缓存

**因此，在使用Hive over HBase，对HBase中的表做统计分析时候，需要特别注意以下几个方面：**

1. 对HBase表进行预分配Region，根据表的数据量估算出一个合理的Region数；

2. rowkey设计上需要注意，尽量使rowkey均匀分布在预分配的N个Region上；

3. 通过set hbase.client.scanner.caching设置合理的扫描器缓存；

4. 关闭mapreduce的推测执行：

set mapred.map.tasks.speculative.execution = false;  
set mapred.reduce.tasks.speculative.execution = false;