创建索引 create index TEST123INDEX on "test123"("f"."c1","f"."c2")[include("f"."c3")]

创建索引，有本地索引，多写入，少夺取；另外是，全局索引，重读，轻写入；

上面的创建索引的命令就是全局索引；另外还引入了覆盖索引的概念（include）,覆盖索引用于避免全表扫描。

# 利用Phoenix为HBase创建二级索引

原创 2015年05月17日 15:10:35

* 标签：
* [Phoenix](http://so.csdn.net/so/search/s.do?q=Phoenix&t=blog) /
* [HBase](http://so.csdn.net/so/search/s.do?q=HBase&t=blog)

# 为什么需要Secondary Index

对于HBase而言，如果想精确地定位到某行记录，唯一的办法是通过rowkey来查询。如果不通过rowkey来查找数据，就必须逐行地比较每一列的值，即全表扫瞄。对于较大的表，全表扫瞄的代价是不可接受的。

但是，很多情况下，需要从多个角度查询数据。例如，在定位某个人的时候，可以通过姓名、身份证号、学籍号等不同的角度来查询，要想把这么多角度的数据都放到rowkey中几乎不可能（业务的灵活性不允许，对rowkey长度的要求也不允许）。

所以，需要secondary index来完成这件事。secondary index的原理很简单，但是如果自己维护的话则会麻烦一些。现在，Phoenix已经提供了对HBase secondary index的支持，下面将说明这样用Phoenix来在HBase中创建二级索引。

# 配置HBase以支持Secondary Index

在每一个RegionServer的hbase-site.xml中加入如下的属性：

<property>

<name>hbase.regionserver.wal.codec</name>

<value>org.apache.hadoop.hbase.regionserver.wal.IndexedWALEditCodec</value>

</property>

<property>

<name>hbase.region.server.rpc.scheduler.factory.class</name>

<value>org.apache.hadoop.hbase.ipc.PhoenixRpcSchedulerFactory</value>

<description>Factory to create the Phoenix RPC Scheduler that uses separate queues for index and metadata updates</description>

</property>

<property>

<name>hbase.rpc.controllerfactory.class</name>

<value>org.apache.hadoop.hbase.ipc.controller.ServerRpcControllerFactory</value>

<description>Factory to create the Phoenix RPC Scheduler that uses separate queues for index and metadata updates</description>

</property>

<property>

<name>hbase.coprocessor.regionserver.classes</name>

<value>org.apache.hadoop.hbase.regionserver.LocalIndexMerger</value>

</property>

12345678910111213141516171819

在每一个Master的hbase-site.xml中加入如下的属性：

<property>

<name>hbase.master.loadbalancer.class</name>

<value>org.apache.phoenix.hbase.index.balancer.IndexLoadBalancer</value>

</property>

<property>

<name>hbase.coprocessor.master.classes</name>

<value>org.apache.phoenix.hbase.index.master.IndexMasterObserver</value>

</property>12345678

# Global Indexing v.s. Local Indexing

## Global Indexing

Global indexing targets read heavy, low write uses cases. With global indexes, all the performance penalties for indexes occur at write time. We intercept the data table updates on write (DELETE, UPSERT VALUES and UPSERT SELECT), build the index update and then sent any necessary updates to all interested index tables. At read time, Phoenix will select the index table to use that will produce the fastest query time and directly scan it just like any other HBase table. By default, unless hinted, an index will not be used for a query that references a column that isn’t part of the index.

## Local Indexing

Local indexing targets write heavy, space constrained use cases. Just like with global indexes, Phoenix will automatically select whether or not to use a local index at query-time. With local indexes, index data and table data co-reside on same server preventing any network overhead during writes. Local indexes can be used even when the query isn’t fully covered (i.e. Phoenix automatically retrieve the columns not in the index through point gets against the data table). Unlike global indexes, all local indexes of a table are stored in a single, separate shared table. At read time when the local index is used, every region must be examined for the data as the exact region location of index data cannot be predetermined. Thus some overhead occurs at read-time.

# Examples

## 为已有的Phoenix表创建secondary index

首先，在Phoenix中创建一个salted table。

create table EXAMPLE (my\_pk varchar(50) not null, M.C0 varchar(50), M.C1 varchar(50), M.C2 varchar(50), M.C3 varchar(50), M.C4 varchar(50), M.C5 varchar(50), M.C6 varchar(50), M.C7 varchar(50),M.C8 varchar(50), M.C9 varchar(50) constraint pk primary key (my\_pk)) salt\_buckets=10;1

这里创建的表名为EXAMPLE，有10个列（列名为{M.C0, M.C1, ···, M.C9}）。

然后，再将一个CSV文件中的数据通过Bulkload的方式导入到该表中。   
该CSV文件中有1000万条记录，在HDFS中的路径为/user/tao/xt-data/data-10m.csv，则导入的命令为

HADOOP\_CLASSPATH=$(hbase classpath) hadoop jar phoenix-4.3.1-client.jar org.apache.phoenix.mapreduce.CsvBulkLoadTool -libjars 3rdlibs/commons-csv-1.0.jar,3rdlibs/joda-time-2.7.jar --table EXAMPLE --input /user/tao/xt-data/data-10m.csv1

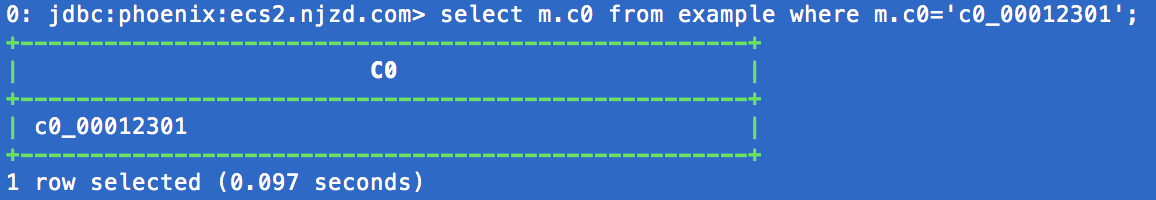
由于导入的过程用到了一些其他的类，所以需要通过-libjars 3rdlibs/commons-csv-1.0.jar,3rdlibs/joda-time-2.7.jar来将相关的Jar包传给这个MapReduce任务。另外，[Bulk CSV Data Loading](http://phoenix.apache.org/bulk_dataload.html)中提到的方法，要求对于Phoenix 4.0+的版本，指定 HADOOP\_CLASSPATH=$(hbase mapredcp)，但是通过实践，发现这样不行，应该用HADOOP\_CLASSPATH=$(hbase classpath)。

在为表EXAMPLE创建secondary index之前，先看看查询一条数据所需的时间：   
可以看到，对名为M.C0的列进行按值查询需要7秒多。

现在，为表EXAMPLE的列M.C0创建Index，如下：

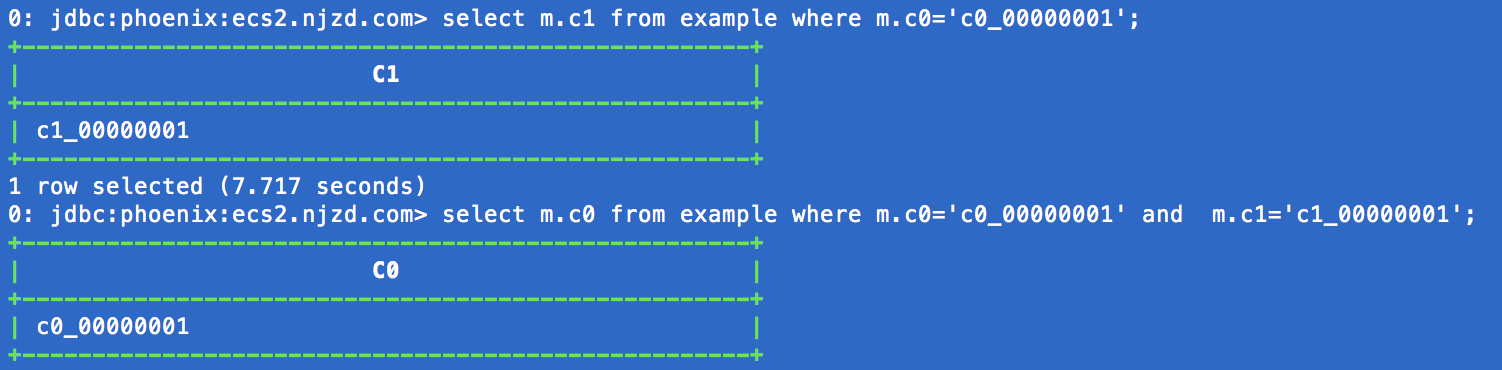
create index my\_index on example (m.c0);1

此时，查看HBase，会发现其中多了一个名为MY\_INDEX的表。而从Phoenix中则看不到该表。

在为EXAMPLE创建了index之后，我们再来进行查询。   
   
这次，查询时间从7秒多降到了0.097秒。

## 确保Query使用Index

By default, a global index will not be used unless all of the columns referenced in the query are contained in the index.

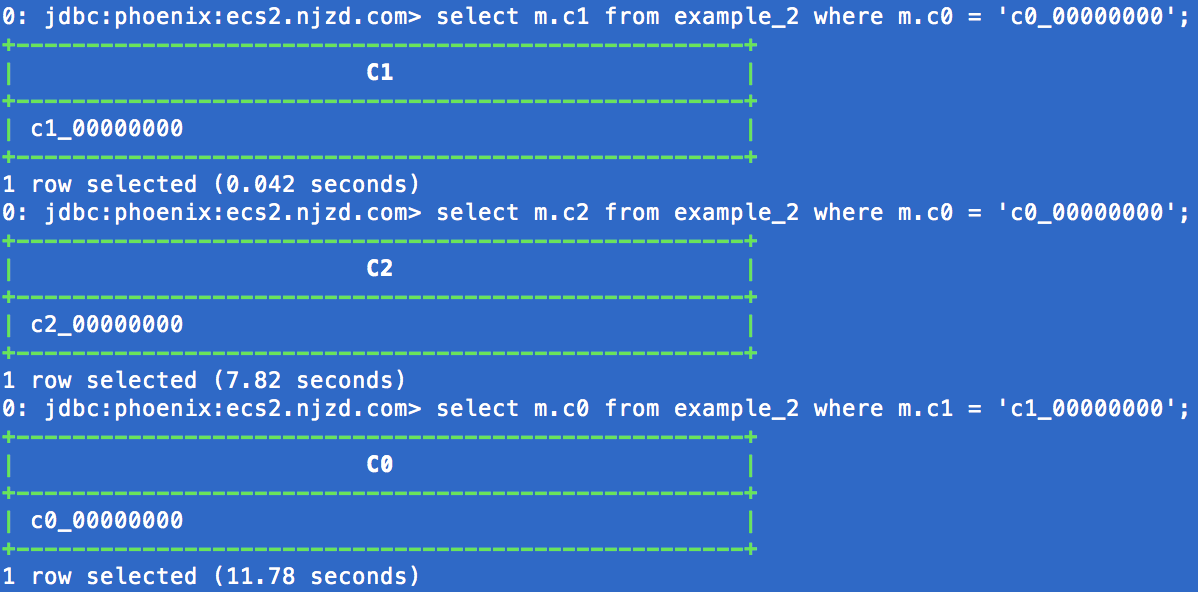
在上例中，由于我们只对M.C0创建了索引，所以如果查询项中包含其他列的话（主键MY\_PK除外），是不会使用index的。此外，如果查询项不包含其他列，但是条件查询语句中包含了其他列（主键MY\_PK除外），也会引发全表扫瞄。如下:   
   
  
  
要让一个查询使用index，有三种方式：   
#1. 创建 convered index；   
#2. 在查询中提示其使用index；   
#3. 创建 local index

### #方式1 - Covered Index

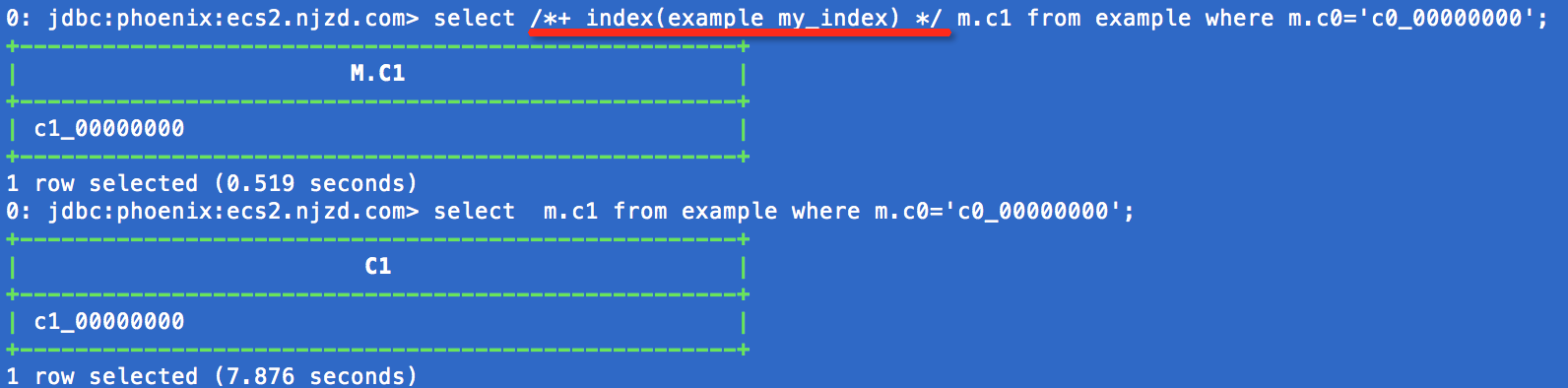
[What is a Covered Index?](http://stackoverflow.com/questions/62137/what-is-a-covered-index)   
[Using Covering Index to Improve Query Performance](https://www.simple-talk.com/sql/learn-sql-server/using-covering-indexes-to-improve-query-performance/)

如果在某次查询中，查询项或者查询条件中包含除被索引列之外的列（主键MY\_PK除外）。默认情况下，该查询会触发full table scan，但是使用covered index则可以避免全表扫描。   
例如，我们按照EXAMPLE的方式创建了另一张表EXAMPLE\_2，表中的数据和表的schema与EXAMPLE都是相同的，不同之处在于EXAMPLE\_2对其M.C1这一列创建了covered index。

create index my\_index\_2 on example\_2 (m.c0) include (m.c1);1

现在，如果查询项中不包含除M.C0和M.C1之外的列，而且查询条件不包含除M.C0之外的列，则可以确保该查询使用Index，如下：   


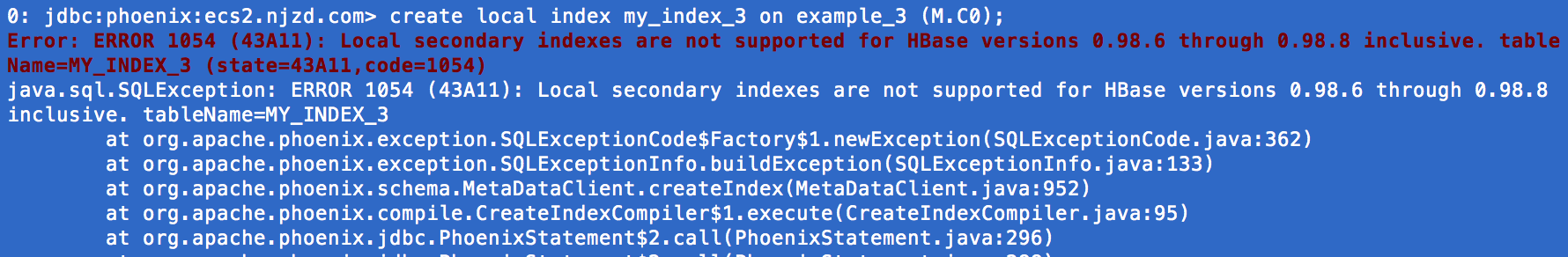
### #方式2 - Hint

在select和column\_name之间加上/\*+ Index(<表名> <index名>)\*/，如下：   


This will cause each data row to be retrieved when the index is traversed to find the missing M.C1 column value. This hint should only be used if you know that the index has good selective (i.e. a small number of table rows have a value of ‘c0\_00000000’ in this example), as otherwise you’ll get better performance by the default behavior of doing a full table scan

### #方式3 - Local Index

Unlike global indexes, local indexes will use an index even when all columns referenced in the query are not contained in the index. This is done by default for local indexes because we know that the table and index data coreside on the same region server thus ensuring the lookup is local.

对于HBase 0.98.6的版本，似乎不支持创建local index，如下：   


## 其他方面

### Functional Index

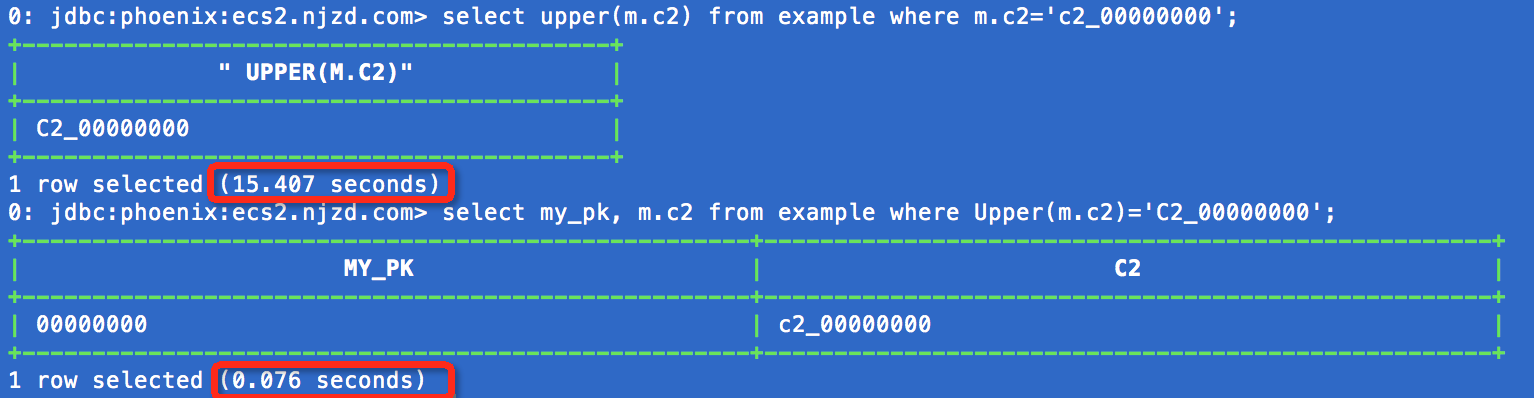
Another useful feature that was introduced in the 4.3 release is functional indexes. Functional indexes allow you to create an index not just on columns, but on an arbitrary expressions. Then when a query uses that expression, the index may be used to retrieve the results instead of the data table. For example, you could create an index on UPPER(FIRST\_NAME||' '||LAST\_NAME) to allow you to do case insensitive searches on the combined first name and last name of a person.

Phoenix supports two types of indexing techniques: global and local indexing. Each are useful in different scenarios and have their own failure profiles and performance characteristics.

下面为表EXAMPLE创建一个Functional Index，如下：

create index index\_upper\_c2 on example (upper(m.c2)) include m.c21

这里，我们实际上为表EXAMPLE又创建了一个名为INDEX\_UPPER\_C2的Index。也就是说，可以为同一张表创建多个Index。



### Index 排序

create index my\_index on example (M.C1 desc, M.C0) include (M.C2);1

### 删除Index

drop index `index-name` on `table-name`1

### 索引表属性

create table和create index都可以将一些属性传递给对应的HBase表，例如：

CREATE INDEX my\_index ON my\_table (v2 DESC, v1) INCLUDE (v3)

SALT\_BUCKETS=10, DATA\_BLOCK\_ENCODING='NONE';12

对于global indexes，如果primary table是salted table，则index会自动地成为salted index。对于local indexes，则不允许指定salt\_buckets。

### Immutable Indexing

For a table in which the data is only written once and never updated in-place, certain optimizations may be made to reduce the write-time overhead for incremental maintenance. This is common with time-series data such as log or event data, where once a row is written, it will never be updated. To take advantage of these optimizations, declare your table as immutable by adding the IMMUTABLE\_ROWS=true property to your DDL statement:

CREATE TABLE my\_table (k VARCHAR PRIMARY KEY, v VARCHAR) IMMUTABLE\_ROWS=true;1

All indexes on a table declared with IMMUTABLE\_ROWS=true are considered immutable (note that by default, tables are considered mutable). For global immutable indexes, the index is maintained entirely on the client-side with the index table being generated as change to the data table occur. Local immutable indexes, on the other hand, are maintained on the server-side. Note that no safeguards are in-place to enforce that a table declared as immutable doesn’t actually mutate data (as that would negate the performance gain achieved). If that was to occur, the index would no longer be in sync with the table.   
  
  
如果创建的表是immutable table，如下：

create table my\_tablek(VARCHAR PRIMARY KEY, v VARCHAR) IMMUTABLE\_ROWS=true1

那么，为该表创建的所有index都是immutable indexes。   
  
  
可以将一个已有的immutable table转变为mutable table，可以通过如下命令：

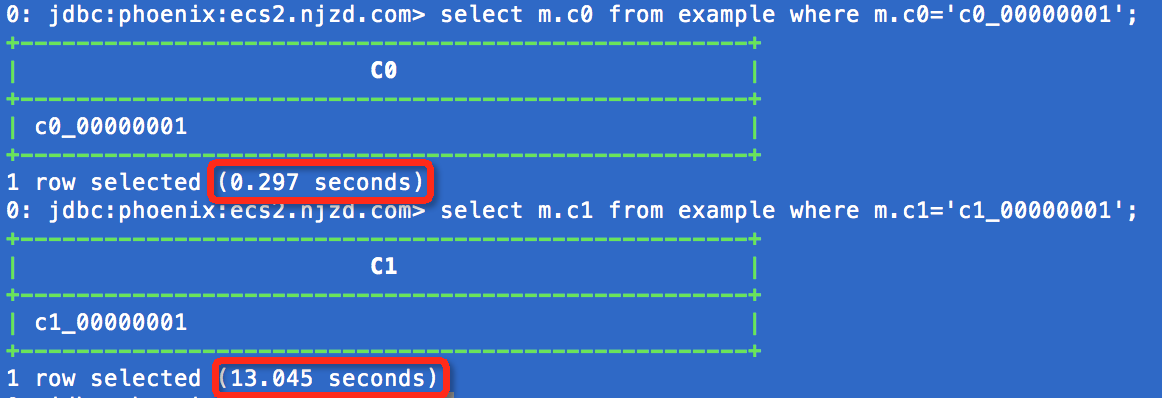
alter table my\_table set IMMUTABLE\_ROWS=false1

# 问题

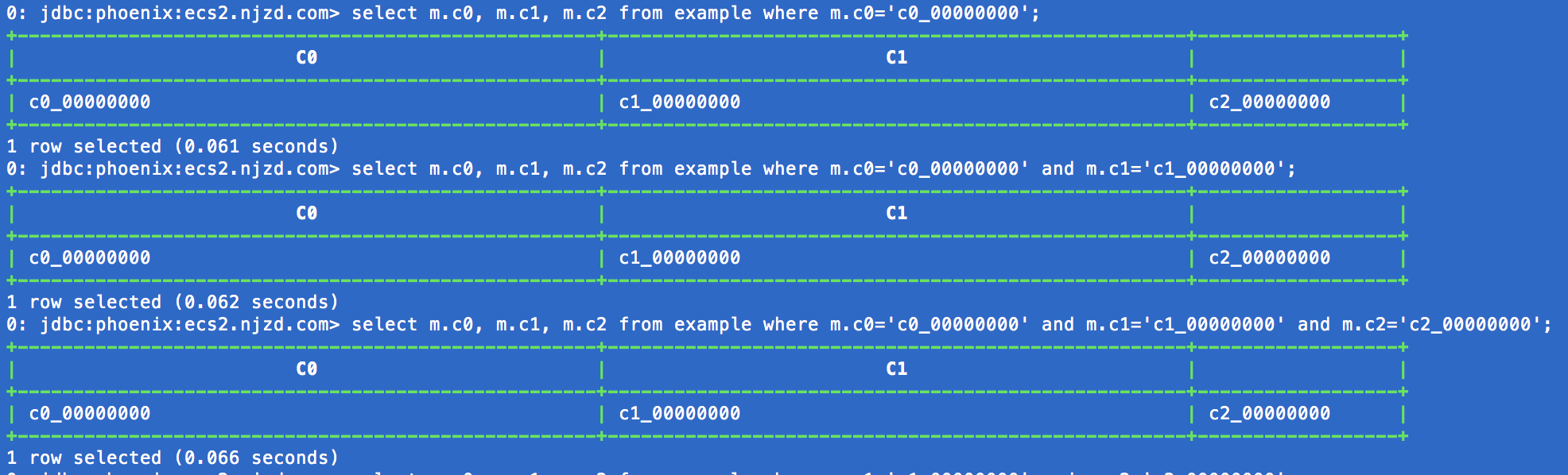
## #1 能不能为多个column建立index（在同一个index table中）？

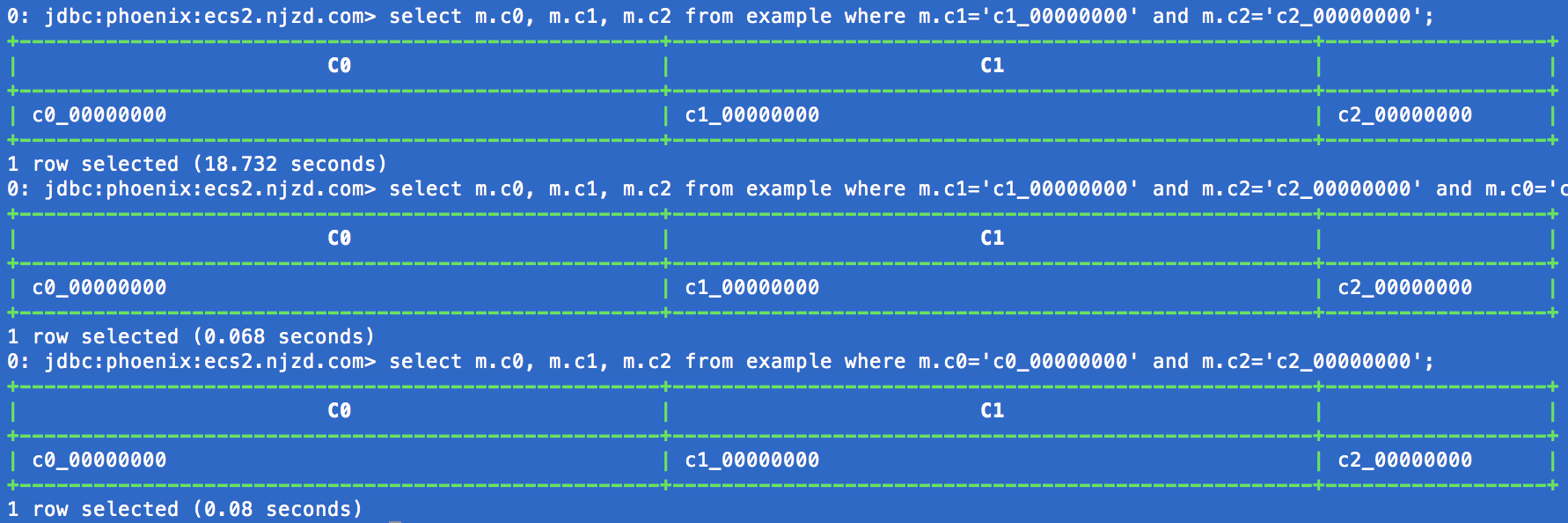
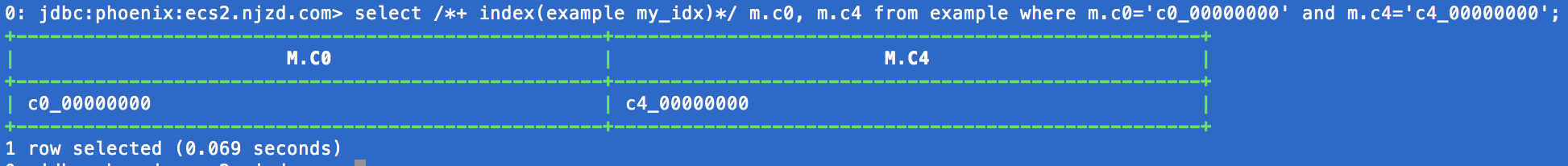
例如，如下命令似乎是可以对两个column创建index：

create index my\_idx on example(m.c0, m.c1)1

但是，似乎只有列m.c0的真正具有索引，列m.c1似乎没有索引：   
   
  
  
答案是：可以为多个column创建索引，但是在查询时要按照索引列的顺序来查询。例如，为M.C0、M.C1和M.C2建立索引：

create index idx on example (m.c0, m.c1, m.c2)1

在查询时，可以同时根据将这3列作为条件，且顺序不限。但是，第一列必须是M.C0。   
   
  
  
这是因为：当为多列建立索引时，rowkey实际上是这些column的组合，并且是按照它们的先后顺序的组合。

   
  
  
如果查询时第一列不是M.C0，那么就要进行full scan，速度会很慢。而如果查询的第一列是M.C0，就可以直接将满足关于M.C0的数据记录找出来。即使后面还有没有被索引的列，也可以很快得到结果，因为满足关于M.C0的结果集已经不大了（如果是这种情况的话），对其再进行一次查询不会是full scan。   


## #2 Bulkload的数据的index能不能自动同步？

维护Index的原理：当对data table执行写入操作时，delete、upsert values和upsert select会被捕获，并据此来更新data table所对应的index。

We intercept the data table updates on write (DELETE, UPSERT VALUES and UPSERT SELECT), build the index update and then sent any necessary updates to all interested index tables

当以bulkload的方式来将数据导入到data table时，会绕开HBase的常规写入路径（client –> buffer –> memstore –> HLog –> StoreFile –> HFile），直接生成最终的HFiles。对于bulkload，对data table的更新能不能被捕获，进而自动维护相应index呢？我们来验证。   
  
  
首先建立一个空的data table：

create table EXAMPLE (PK varchar primary key, M.C0 varchar, M.C1 varchar, M.C2 varchar, M.C3 varchar, M.C4 varchar, M.C5 varchar, M.C6 varchar, M.C7 varchar, M.C8 varchar, M.C9 varchar) salt\_buckets = 201

再为其创建2个index：

create index IDX\_C0 on EXAMPLE(M.C0);

create index IDX\_C1 on EXAMPLE(M.C1);