第一部分 概述

背景

Apache Kudu是由Cloudera开源的存储引擎，可以同时提供低延迟的随机读写和高效的数据分析能力。Kudu支持水平扩展，使用Raft协议进行一致性保证，并且与Cloudera Impala和Apache Spark等当前流行的大数据查询和分析工具结合紧密。

近两年，KUDU 在大数据平台的应用越来越广泛。在阿里、小米、网易等公司的大数据架构中，KUDU 都有着不可替代的地位。

现在提起大数据存储，我们能想到的技术有很多，比如HDFS，以及在HDFS上的列式存储技术Apache Parquet，Apache ORC，还有以KV形式存储半结构化数据的Apache HBase和Apache Cassandra等等。既然有了如此多的存储技术，Cloudera公司为什么要开发出一款全新的存储引擎Kudu呢？

基于HDFS的存储技术：

数据分析：Parquet，具有高吞吐量连续读取数据的能力

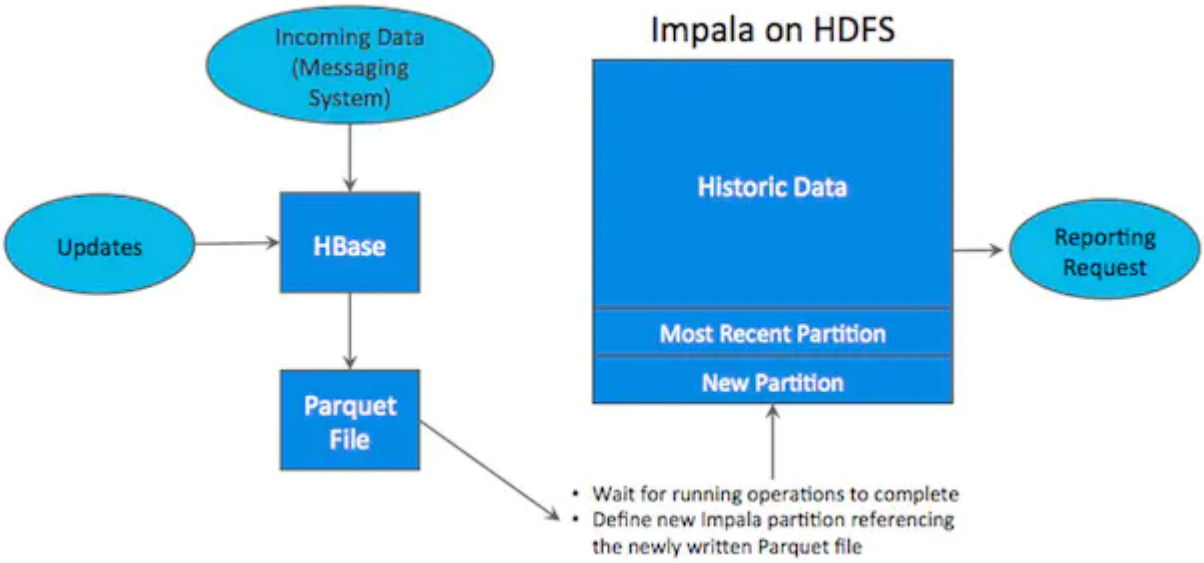
实时读写：HBase和Cassandra等技术适用于低延迟的随机读写场景

在 KUDU 之前，大数据主要以两种方式存储：

- **静态数据**：以 HDFS 引擎作为存储引擎，适用于高吞吐量的离线大数据分析场景。这类存储的局限性是数据无法进行随机的读写。
- **动态数据**：以 HBase、Cassandra 作为存储引擎，适用于大数据随机读写场景。这类存储的局限性是批量读取吞吐量远不如 HDFS，不适用于批量数据分析的场景。

所以现在的企业中，经常会存储两套数据分别用于实时读写与数据分析，

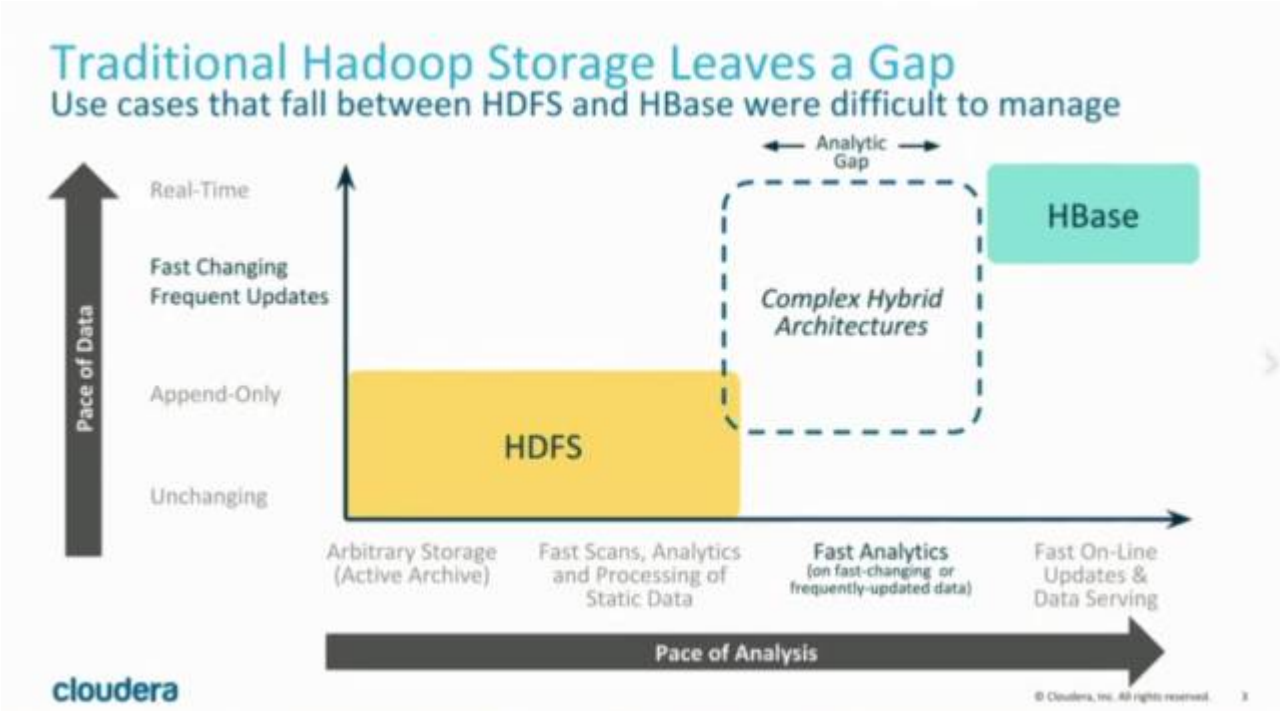
先将数据写入HBase中，再定期通过ETL到Parquet进行数据同步。



但是这样做有很多缺点：

- 用户需要在两套系统间编写和维护复杂的ETL逻辑。结构复杂，维护成本高。
- 时效性较差。因为ETL通常是一个小时、几个小时甚至是一天一次，那么可供分析的数据就需要一个小时至一天的时间后才进入到可用状态，也就是说从数据到达到可被分析之间是会存在一个较为明显的“空档期”的。
- 更新需求难以满足。在实际情况中可能会有一些对已经写入的数据的更新需求，这种情况往往需要对历史数据进行更新，而对Parquet这种静态数据集的更新操作，代价是非常昂贵的。
- 存储资源浪费。两套存储系统意味着占用的磁盘资源翻倍了，造成了成本的提升。

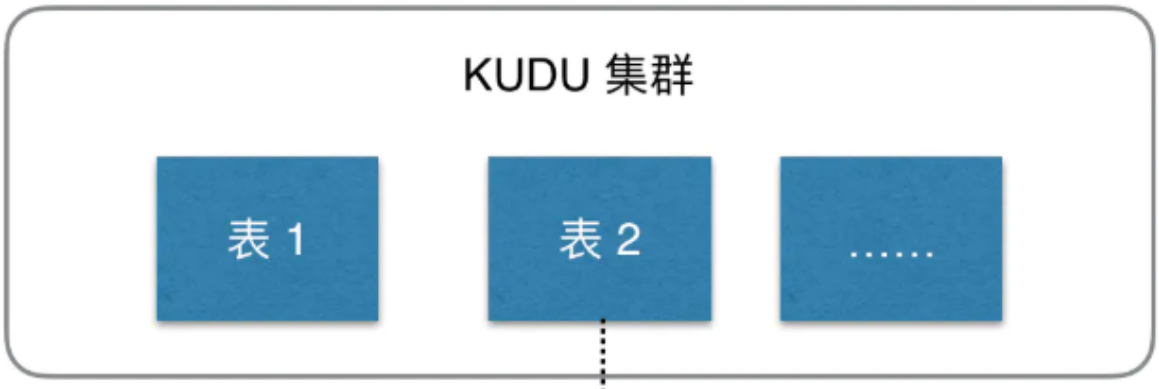
我们知道，基于HDFS的存储技术，比如Parquet，具有高吞吐量连续读取数据的能力；而HBase和Cassandra等技术适用于低延迟的随机读写场景，那么有没有一种技术可以同时具备这两种优点呢？Kudu提供了一种“happy medium”的选择：



Kudu不但提供了行级的插入、更新、删除API，同时也提供了接近Parquet性能的批量扫描操作。使用同一份存储，既可以进行随机读写，也可以满足数据分析的要求。

数据模型

KUDU 的数据模型与传统的关系型数据库类似，一个 KUDU 集群由多个**表**组成，每个表由多个**字段**组成，一个表必须指定一个由若干个（>=1）字段组成的**主键**，如下图：



Twitter Firehose Table

tweet_id	user_name	created_at	text
INT64	STRING	TIMESTAMP	STRING
23059873	newsycbot	1442865158	Visual Explanation of the Raft Consensus Algorithm http://bit.ly/1DOUac0 (cmts http://bit.ly/1HKmjfc)
22309487	RideImpala	1442828307	Introducing the Ibis project: for the Python experience at Hadoop Scale
23059861	fastly	1442865156	Missed July's SF @papers_we_love? You can now watch @el_bhs talk about @google's globally-distributed database: http://fastly.us/1eVz8MM
23010982	llvmorg	1442865155	LLVM 3.7 is out! Get it while it's HOT! http://llvm.org/releases/download.html#3.7.0

从用户角度来看，

Kudu是一种存储结构化数据表的存储系统。

在一个Kudu集群中可以定义任意数量的table，每个table都需要预先定义好schema。

每个table的列数是确定的，每一列都需要有名字和类型，每个表中可以把其中一列或多列定义为主键。

这么看来，Kudu更像关系型数据库，而不是像HBase、Cassandra和MongoDB这些NoSQL数据库。不过Kudu目前还不能像关系型数据一样支持二级索引。 Kudu使用确定的列类型，字段是强类型的，而不是类似于NoSQL的“everything is byte”。这可以带来两点好处：

- 确定的列类型使Kudu可以进行类型特有的编码，节省空间。
- 可以提供 SQL-like 元数据给其他上层查询工具，比如BI工具。

- KUDU 的使用场景是 OLAP 分析，有一个数据类型对下游的分析工具也更加友好。

用户可以使用 Insert，Update和Delete API对表进行写操作。不论使用哪种API，都必须指定主键。但批量的删除和更新操作需要依赖更高层次的组件（比如Impala、Spark）。Kudu目前还不支持多行事务。 而在读操作方面，Kudu只提供了Scan操作来获取数据。用户可以通过指定过滤条件来获取自己想要读取的数据，但目前只提供了两种类型的过滤条件：主键范围和列值与常数的比较。由于Kudu在硬盘中的数据采用列式存储，所以只扫描需要的列将极大地提高读取性能。

一致性模型

内部事务隔离

Kudu为用户提供了两种一致性模型。默认的一致性模型是snapshot consistency。这种一致性模型保证用户每次读取出来的都是一个可用的快照，但这种一致性模型只能保证单个client可以看到最新的数据，但不能保证多个client每次取出的都是最新的数据。

另一种一致性模型external consistency可以在多个client之间保证每次取到的都是最新数据，但是Kudu没有提供默认的实现，需要用户做一些额外工作。

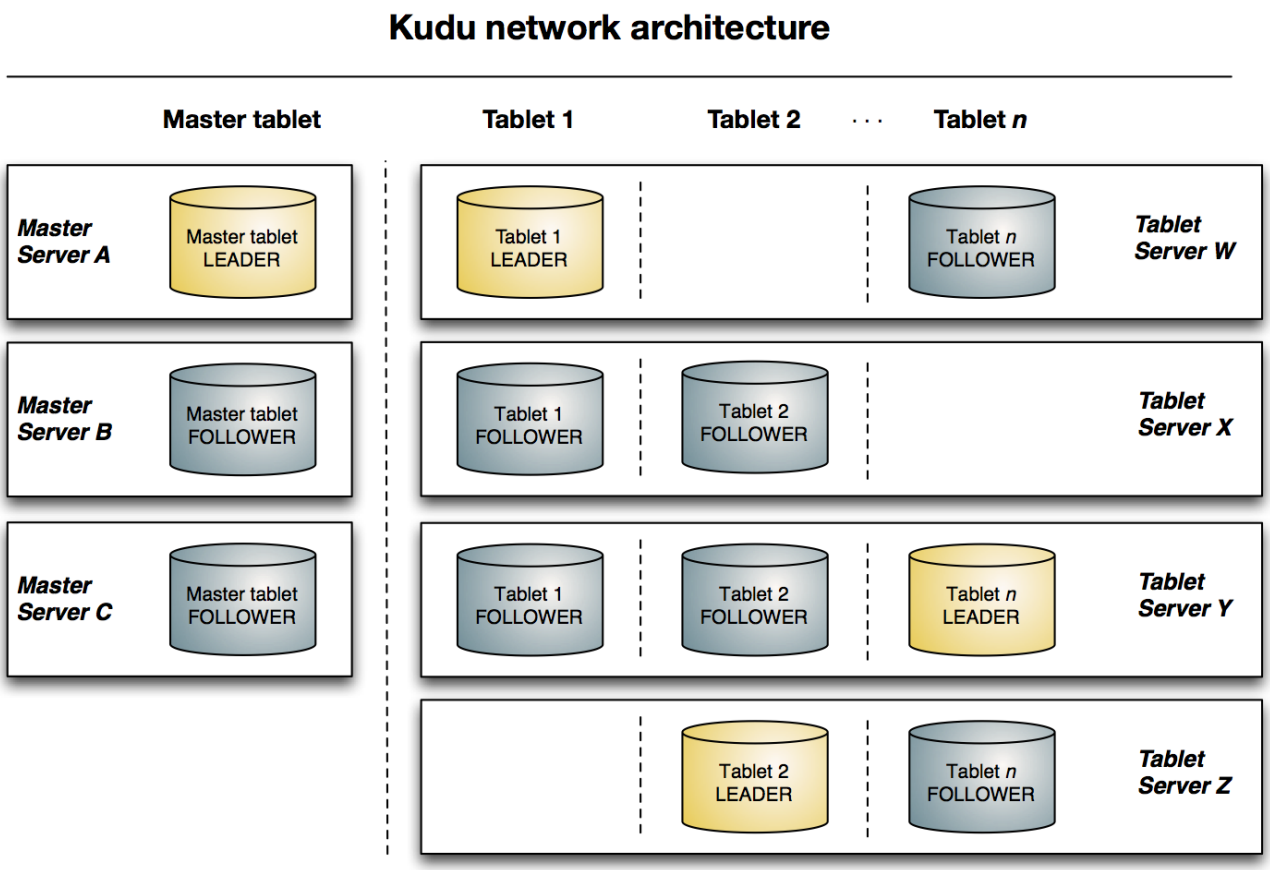
为了实现external consistency，Kudu提供了两种方式：

- 在client之间传播timestamp token。在一个client完成一次写入后，会得到一个timestamp token，然后这个client把这个token传播到其他client，这样其他client就可以通过token取到最新数据了。不过这个方式的复杂度很高。
- 通过commit-wait方式，这有些类似于Google的Spanner。但是目前基于NTP的commit-wait方式延迟实在有点高。不过Kudu相信，随着Spanner的出现，未来几年内基于real-time clock的技术将会逐渐成熟。

第二部分 Kudu的架构

与HDFS和HBase相似，Kudu使用单个的Master节点，用来管理集群的元数据，并且使用任意数量的Tablet Server节点用来存储实际数据。可以部署多个Master节点来提高容错性。

Master



Kudu的master节点负责整个集群的元数据管理和服务协调。它承担着以下功能：

- 作为catalog manager，master节点管理着集群中所有table和tablet的schema及一些其他的元数据。
- 作为cluster coordinator，master节点追踪着所有server节点是否存活，并且当server节点挂掉后协调数据的重新分布。
- 作为tablet directory，master跟踪每个tablet的位置。

Catalog Manager

Kudu的master节点会持有一个单tablet的table——catalog table，但是用户是不能直接访问的。master将内部的catalog信息写入该tablet，并且将整个catalog的信息缓存到内存中。随着现在商用服务器上的内存越来越大，并且元数据信息占用的空间其实并不大，所以master不容易存在性能瓶颈。catalog table保存了所有table的schema的版本以及table的状态（创建、运行、删除等）。

Cluster Coordination

Kudu集群中的每个tablet server都需要配置master的主机名列表。当集群启动时，tablet server会向master注册，并发送所有tablet的信息。tablet server第一次向master发送信息时会发送所有tablet的全量信息，后续每次发送则只会发送增量信息，仅包含新创建、删除或修改的tablet的信息。 作为cluster coordination，master只是集群状态的观察者。对于tablet server中tablet的副本位置、Raft配置和schema版本等信息的控制和修改由tablet server自身完成。master只需要下发命令，tablet server执行成功后会自动上报处理的结果。

Tablet Directory

因为master上缓存了集群的元数据，所以client读写数据的时候，肯定是要通过master才能获取到tablet的位置等信息。但是如果每次读写都要通过master节点的话，那master就会变成这个集群的性能瓶颈，所以client会在本地缓存一份它需要访问的tablet的位置信息，这样就不用每次读写都从master中获取。 因为tablet的位置可能也会发生变化（比如某个tablet server节点crash掉了），所以当tablet的位置发生变化的时候，client会收到相应的通知，然后再去master上获取一份新的元数据信息。

Table

在数据存储方面，Kudu选择完全由自己实现，而没有借助于已有的开源方案。tablet存储主要想要实现的目标为：

- 快速的列扫描
- 低延迟的随机读写
- 一致性的性能

RowSets

在Kudu中，tablet被细分为更小的单元，叫做RowSets。一些RowSet仅存在于内存中，被称为MemRowSets，而另一些则同时使用内存和硬盘，被称为DiskRowSets。任何一行未被删除的数据都只能存在于一个RowSet中。 无论任何时候，一个tablet仅有一个MemRowSet用来保存最新插入的数据，并且有一个后台线程会定期把内存中的数据flush到硬盘上。 当一个MemRowSet被flush到硬盘上以后，一个新的MemRowSet会替代它。而原有的MemRowSet会变成一到多个DiskRowSet。flush操作是完全同步进行的，在进行flush时，client同样可以进行读写操作。

MemRowSet

MemRowSets是一个可以被并发访问并进行过锁优化的B-tree，主要是基于MassTree来设计的，但存在几点不同：

- Kudu并不支持直接删除操作，由于使用了MVCC，所以在Kudu中删除操作其实是插入一条标志着删除的数据，这样就可以推迟删除操作。
- 类似删除操作，Kudu也不支持原地更新操作。
- 将tree的leaf链接起来，就像B+-tree。这一步关键的操作可以明显地提升scan操作的性能。
- 没有实现字典树（trie树），而是只用了单个tree，因为Kudu并不适用于极高的随机读写的场景。

与Kudu中其他模块中的数据结构不同，MemRowSet中的数据使用行式存储。因为数据都在内存中，所以性能也是可以接受的，而且Kudu对在MemRowSet中的数据结构进行了一定的优化。

DiskRowSet

当MemRowSet被flush到硬盘上，就变成了DiskRowSet。当MemRowSet被flush到硬盘的时候，每32M就会形成一个新的DiskRowSet，这主要是为了保证每个DiskRowSet不会太大，便于后续的增量**compaction**操作。Kudu通过将数据分为base data和delta data，来实现数据的更新操作。Kudu会将数据按列存储，数据被切分成多个page，并使用B-tree进行索引。除了用户写入的数据，Kudu还会将主键索引存入一个列中，并且提供布隆过滤器来进行高效查找。

Compaction

为了提高查询性能，Kudu会定期进行compaction操作，合并delta data与base data，对标记了删除的数据进行删除，并且会合并一些DiskRowSet。

分区

选择分区策略需要理解**数据模型**和**表的预期工作负载**：

- 对于写量大的工作负载，重要的是要设计分区，使写分散在各个tablet上，以避免单个tablet超载。

- 对于涉及许多短扫描的工作负载(其中联系远程服务器的开销占主导地位)，如果扫描的所有数据都位于同一块tablet上，则可以提高性能。

理解这些基本的权衡是设计有效分区模式的核心。

没有默认分区 在创建表时，Kudu不提供默认的分区策略。建议预期具有繁重读写工作负载的新表至少拥有与tablet服务器相同的tablet。

和许多分布式存储系统一样，Kudu的table是水平分区的。BigTable只提供了range分区，Cassandra只提供hash分区，而Kudu同时提供了这两种分区方式，使分区较为灵活。当用户创建一个table时，可以同时指定table的的partition schema，partition schema会将primary key映射为partition key。一个partition schema包括0到多个hash-partitioning规则和一个range-partitioning规则。通过灵活地组合各种partition规则，用户可以创造适用于自己业务场景的分区方式。

第三部分 安装和运行

安装前提和准备

硬件:

- 一台或者多台机器跑kudu-master。建议跑一个master(无容错机制)、三个master(允许一个节点运行出错)或者五个master(允许两个节点出错)。
- 一台或者多台机器跑kudu-tserver。当需要使用副本，至少需要三个节点运行kudu-tserver服务。

操作系统(主要是linux系统，windows系统不支持):

- RHEL 6, RHEL 7, CentOS 6, CentOS 7, Ubuntu 14.04 (Trusty), Ubuntu 16.04 (Xenial), Debian 8 (Jessie), or SLES 12.
- 内核和文件系统支持 hole punching 选项。
- ntp服务。
- xfs or ext4 formatted drives

存储:

- 尽量使用固态存储，将显著提高kudu性能。

管理

- 如果你使用的是CDH，需要Cloudera Manager 5.4.3及以上的版本。

环境说明

- os：CentOS Linux release 7.6.1810 (Core)
- hdp-1 hdp-2 hpd-3 三台机器 hdp-1启动Master hdp-1,hdp-2,hdp-3启动tserver

安装ntp服务

每个节点执行:

```
yum -y install ntp
```

- 注释掉以下四行:

```
#server 0.centos.pool.ntp.org iburst
#server 1.centos.pool.ntp.org iburst
#server 2.centos.pool.ntp.org iburst
#server 3.centos.pool.ntp.org iburst
```

- 修改hdp-2 192.168.81.130节点上的配置文件

```
vi /etc/ntp.conf
```

加入如下内容

```
restrict 192.168.81.0 mask 255.255.255.0 notrap nomodify   # 给192.168.81.0网段，子网掩码为255.255.255.0的局域网机的机器有同步时间的权限
server  192.168.81.130   prefer                             # prefer代表优先使用此ip做同步
server 127.127.1.0                                          # 当所有服务器都不能使用时，使用本机作为同步服务器
fudge   127.127.1.0 stratum 10
```

- 修改192.168.81.129和192.16881.131节点上的配置文件

```
vi /etc/ntp.conf
```

加入以下内容

```
server  192.168.81.130 prefer
server  127.127.1.0
fudge   127.127.1.0 stratum 10
```

启动NTP服务

```
service ntpd start

chkconfig ntpd on
```

检验

检查ntp服务是否成功输入：ntpstat

输出如下则启动成功并且同步已完成

```
synchronised to local net at stratum 11
time correct to within 11 ms
polling server every 64 s
```

 # /etc/init.d/ntpd start 各个节点检查启动成功，否则启动kudu相关服务会报错

时钟同步，kudu对时间要求很精准

```
[root@hdp-1 kudu]#  ntpdate -u ntp.api.bz
 1 Sep 16:37:25 ntpdate[13780]: adjust time server 114.118.7.161 offset -0.011283 sec
[root@hdp-1 kudu]# systemctl status ntpd.service
● ntpd.service - Network Time Service
   Loaded: loaded (/usr/lib/systemd/system/ntpd.service; enabled; vendor preset: disabled)
   Active: active (running) since Tue 2020-09-01 02:07:39 CST; 14h ago
 Main PID: 13476 (ntpd)
   CGroup: /system.slice/ntpd.service
           └─13476 /usr/sbin/ntpd -u ntp:ntp -g

Sep 01 02:07:39 hdp-1 ntpd[13476]: Listen normally on 3 ens33 192.168.81.129 UDP 123
Sep 01 02:07:39 hdp-1 ntpd[13476]: Listen normally on 4 lo ::1 UDP 123
Sep 01 02:07:39 hdp-1 ntpd[13476]: Listen normally on 5 ens33 fe80::20c:29ff:fef6:11ed UDP
123
Sep 01 02:07:39 hdp-1 ntpd[13476]: Listening on routing socket on fd #22 for interface
updates
Sep 01 02:07:50 hdp-1 ntpd[13476]: 0.0.0.0 c016 06 restart
Sep 01 02:07:50 hdp-1 ntpd[13476]: 0.0.0.0 c012 02 freq_set kernel 0.000 PPM
Sep 01 02:07:50 hdp-1 ntpd[13476]: 0.0.0.0 c011 01 freq_not_set
Sep 01 02:07:57 hdp-1 ntpd[13476]: 0.0.0.0 c61c 0c clock_step +51374.863232 s
Sep 01 16:24:12 hdp-1 ntpd[13476]: 0.0.0.0 c614 04 freq_mode
Sep 01 16:24:13 hdp-1 ntpd[13476]: 0.0.0.0 c618 08 no_sys_peer
```

配置Yum的Repository

- 在使用 yum来安装kudu的时候，由于kudu不是yum的常规组建，直接安装会找不到kudu，所以第一步需要将kudu的repo文件下载并放置到合适的位置。

1. 下载kudu的repo文件 下载repo文件：`wget http://archive.cloudera.com/kudu/redhat/7/x86_64/kudu/cloudera-kudu.repo`

   可以直接从地址下载

2. 将下载的repo文件放置到/etc/yum.repos.d/目录下

```
sudo mv cloudera-kudu.repo mv /etc/yum.repos.d/
```

安装kudu

- 安装，在每个节点上执行

  ```
  # yum install kudu kudu-master kudu-client0 kudu-client-devel -y
  ```

- 配置并启动kudu
- hdp-2: master hdp-1,hdp-2,hdp-3 slaver

  安装完成，在/etc/kudu/conf目录下有两个文件：master.gflagfile和tserver.gflagfile

  使用192.168.81.130作为kudu-master，192.168.81.129、192.168.20.130和192.168.81.131作为kudu-tserver节点

所以192.168.81.130节点需要修改master.gflagfile和tserver.gflagfile文件，而192.168.81.129和192.168.20.131只需要修改tserver.gflagfile文件

- 修改kudu-master启动配置 hdp-2节点

  ```
  vi /etc/default/kudu-master
  ```

  修改以下内容，主要是修改ip：

  ```
  export FLAGS_rpc_bind_addresses=192.168.81.130:7051
  ```

- 修改每个节点的kudu-tserver启动配置

  ```
  vi /etc/default/kudu-tserver
  ```

  修改以下内容，主要是修改ip为当前节点ip

  ```
  export FLAGS_rpc_bind_addresses=192.168.81.130:7050
  ```

- master.gflagfile的配置修改

  ```
  --fromenv=rpc_bind_addresses
  --fromenv=log_dir

  --fs_wal_dir=/var/lib/kudu/master
  --fs_data_dirs=/var/lib/kudu/master
  -unlock_unsafe_flags=true
  -allow_unsafe_replication_factor=true
  -default_num_replicas=1         # 此参数可以调整备份数量，默认为3
  ```

- tserver.gflagfile 的配置修改

  ```
  # Do not modify these two lines. If you wish to change these variables,
  # modify them in /etc/default/kudu-tserver.
  --fromenv=rpc_bind_addresses
  --fromenv=log_dir

  --fs_wal_dir=/var/lib/kudu/tserver
  --fs_data_dirs=/var/lib/kudu/tserver
  --tserver_master_addrs=hdp-2:7051

  -unlock_unsafe_flags=true
  -allow_unsafe_replication_factor=true
  -default_num_replicas=1
  --tserver_master_addrs=192.168.81.130:7051   # 此参数指定master
  ```

  注意，这里的-tserver_master_addrs指明了集群中master的地址，指向同一个master的tserver形成了一个kudu集群

- 创建master.gflagfile和tserver.gflagfile文件中指定的目录，并将所有者更改为kudu，执行如下命令：

```
mkdir -p /var/lib/kudu/master /var/lib/kudu/tserver

chown -R kudu:kudu /var/lib/kudu/
```

- 修改 /etc/security/limits.d/20-nproc.conf 文件，解除kudu用户的线程限制,注意：20可能不同，根据自己的来修改

  ```
  vi /etc/security/limits.d/20-nproc.conf
  ```

  加入以下两行内容

  ```
  kudu       soft    nproc     unlimited
  impala     soft    nproc     unlimited
  ```

- 启动kudu

  master节点(hdp-2 192.168.81.130)执行：

  ```
  service kudu-master start
  service kudu-tserver start
  ```

  192.168.81.129和192.168.81.131执行：

  ```
  service kudu-tserver start
  ```

问题1：kudu-master启动失败

查看/var/log/kudu/err

发现时间问题，解决方案，重启ntpd service ntpd restart

然后重启kudu-master service kudu-master restart

第四部分 KuDu常用Api(java)

1.首先创建一个maven工程，添加一下依赖

```
<dependency>
      <groupId>org.apache.kudu</groupId>
      <artifactId>kudu-client</artifactId>
      <version>1.4.0</version>
 </dependency>
```

2.创建表

思路：

kudu建表几个必须：

（1）必须指定表连接到的master节点主机名

（2）必须定义schema

（3）必须指定副本数量、分区策略和数量

```
package lagou;

import org.apache.kudu.ColumnSchema;
import org.apache.kudu.Schema;
import org.apache.kudu.Type;
import org.apache.kudu.client.CreateTableOptions;
import org.apache.kudu.client.KuduClient;
import org.apache.kudu.client.KuduException;

import java.util.ArrayList;

public class createTableDemo {
    /**
     *  (1) 必须指定表连接到的master节点主机名
     *
     *  (2) 必须定义schema,声明的每一个字段必须显式的说明是否是主键
     *
     *  (3) 必须指定副本数量、分区策略和数量
     */
    public static void main(String[] args) {
        //client
        String masterAddresses = "hdp-2";
        KuduClient.KuduClientBuilder kuduClientBuilder = new
KuduClient.KuduClientBuilder(masterAddresses);
        KuduClient client = kuduClientBuilder.build();

        String tableName = "student";
        //id int pkey    name string

        //指定每一列的信息
        ArrayList<ColumnSchema> columnSchemas = new ArrayList<ColumnSchema>();
        ColumnSchema id = new ColumnSchema.ColumnSchemaBuilder("id",
Type.INT32).key(true).build();
        ColumnSchema name = new ColumnSchema.ColumnSchemaBuilder("name",
Type.STRING).key(false).build();
        columnSchemas.add(id);
        columnSchemas.add(name);
        Schema schema = new Schema(columnSchemas);

        CreateTableOptions options = new CreateTableOptions();
        //设定当前的副本数量为1
        options.setNumReplicas(1);
        ArrayList<String> colrule = new ArrayList<String>();
        colrule.add("id");
        options.addHashPartitions(colrule,3);
```

```
        try {
            client.createTable(tableName,schema,options);
        } catch (KuduException e) {
            e.printStackTrace();
        } finally {
            try {
                client.close();
            } catch (KuduException e) {
                e.printStackTrace();
            }
        }
    }

}
```

3.删除表

client.deleteTable

```
package lagou;

import org.apache.kudu.client.KuduClient;
import org.apache.kudu.client.KuduException;

public class deleteTableDemo {

    public static void main(String[] args) {
        //master地址
        String masterAddr = "192.168.56.56";

        KuduClient client = new KuduClient.KuduClientBuilder(masterAddr)
                .defaultSocketReadTimeoutMs(6000).build();

        try {
            client.deleteTable("student");
        } catch (KuduException e) {
            e.printStackTrace();
        }finally {
            try {
                client.close();
            } catch (KuduException e) {
                e.printStackTrace();
            }
        }

    }
}
```

4.插入数据

思路:

1、获取客户端

2、打开一张表

3、创建会话

4、设置刷新模式

5、获取插入实例

6、声明带插入数据

7、刷入数据

8、应用插入实例

9、关闭会话

1、AUTO_FLUSH_SYNC（默认），意思是调用 KuduSession.apply() 方法后，客户端会在当数据刷新到服务器后再返回，这种情况就不能批量插入数据，调用 KuduSession.flush() 方法不会起任何作用，应为此时缓冲区数据已经被刷新到了服务器。

2、AUTO_FLUSH_BACKGROUND，意思是调用 KuduSession.apply() 方法后，客户端会立即返回，但是写入将在后台发送，可能与来自同一会话的其他写入一起进行批处理。如果没有足够的缓冲空间，KuduSession.apply()会阻塞，缓冲空间不可用。因为写入操作是在后台应用进行的的，因此任何错误都将存储在一个会话本地缓冲区中。注意：这个模式可能会导致数据插入是乱序的，这是因为在这种模式下，多个写操作可以并发地发送到服务器。即此处为 kudu 自身的一个 bug,KUDU-1767 已经说明。

3、MANUAL_FLUSH,意思是调用 KuduSession.apply() 方法后，会返回的非常快,但是写操作不会发送，直到用户使用flush()函数，如果缓冲区超过了配置的空间限制，KuduSession.apply()函数会返回一个错误。

```
package lagou;

import org.apache.kudu.client.*;

public class insertDemo {
    public static void main(String[] args) {
        String masterAddr = "hdp-2";

        KuduClient.KuduClientBuilder builder = new KuduClient.KuduClientBuilder(masterAddr);
        builder.defaultSocketReadTimeoutMs(6000);
        KuduClient client = builder.build();

        try {
            KuduTable stuTable = client.openTable("student");
            KuduSession kuduSession = client.newSession();
            kuduSession.setFlushMode(SessionConfiguration.FlushMode.MANUAL_FLUSH);
            Insert insert = stuTable.newInsert();
            insert.getRow().addInt("id",1);
            insert.getRow().addString("name","lucas");
            kuduSession.flush();
            kuduSession.apply(insert);
            kuduSession.close();
```

```
        } catch (KuduException e) {
            e.printStackTrace();
        }
    }
}
```

5.查询数据

kudu查询数据用scanner

思路:

1、获取client

2、获取Scanner

3、从Scanner中循环遍历数据

```
package lagou;

import org.apache.kudu.client.*;

public class selectDemo {
    public static void main(String[] args) {
        KuduClient.KuduClientBuilder clientBuilder = new KuduClient.KuduClientBuilder("hdp-
2");
        KuduClient client = clientBuilder.build();

        try {
            KuduTable stuTable = client.openTable("student");
            KuduScanner scanner = client.newScannerBuilder(stuTable).build();
            while(scanner.hasMoreRows()) {
                for(RowResult result : scanner.nextRows()) {
                    int id = result.getInt("id");
                    String name = result.getString("name");
                    System.out.println(id + name);
                }
            }
        } catch (KuduException e) {
            e.printStackTrace();
        } finally {
            try {
                client.close();
            } catch (KuduException e) {
                e.printStackTrace();
            }
        }
    }
}
```

6.更改表数据

和插入数据类似

```
package lagou;

import org.apache.kudu.client.*;

public class updateDemo {
    public static void main(String[] args) {
        KuduClient client = new KuduClient.KuduClientBuilder("hdp-2").build();
        try {
            KuduTable stuTable = client.openTable("student");
            KuduSession kuduSession = client.newSession();
            kuduSession.setFlushMode(SessionConfiguration.FlushMode.MANUAL_FLUSH);
            Update update = stuTable.newUpdate();
            update.getRow().addInt("id",1);
            update.getRow().addString("name","xiaoming");
            kuduSession.apply(update);
            kuduSession.close();
        } catch (KuduException e) {
            e.printStackTrace();
        } finally {
            try {
                client.close();
            } catch (KuduException e) {
                e.printStackTrace();
            }
        }
    }
}
```

7.删除指定行

通过封装类Delete

```
package lagou;

import org.apache.kudu.client.*;

public class deleteDemo {
    public static void main(String[] args) {
        KuduClient client = new KuduClient.KuduClientBuilder("hdp-2").build();
        try {
            KuduTable stuTable = client.openTable("student");
            KuduSession kuduSession = client.newSession();
            kuduSession.setFlushMode(SessionConfiguration.FlushMode.MANUAL_FLUSH);

            Delete delete = stuTable.newDelete();
```

```
            PartialRow row = delete.getRow();
            row.addInt("id",1);

            kuduSession.flush();
            kuduSession.apply(delete);

            kuduSession.close();

        } catch (KuduException e) {
            e.printStackTrace();
        } finally {
            try {
                client.close();
            } catch (KuduException e) {
                e.printStackTrace();
            }
        }
    }
}
```

第五部分 Flink下沉数据到kudu

思路:

自定义下沉器： extends RichSinkFunction

1、依赖:

```
                <dependency>
            <groupId>org.apache.kudu</groupId>
            <artifactId>kudu-client</artifactId>
            <version>1.5.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-java -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-java</artifactId>
            <version>1.11.1</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-streaming-java -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-streaming-java_2.12</artifactId>
            <version>1.11.1</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-clients -->
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-clients_2.12</artifactId>
            <version>1.11.1</version>
        </dependency>
```

```
        <dependency>
            <groupId>org.apache.flink</groupId>
            <artifactId>flink-table-api-java-bridge_2.12</artifactId>
            <version>1.11.1</version>
        </dependency>
```

2、数据源：

```
new UserInfo("001", "Jack", 18),
new UserInfo("002", "Rose", 20),
new UserInfo("003", "Cris", 22),
new UserInfo("004", "Lily", 19),
new UserInfo("005", "Lucy", 21),
new UserInfo("006", "Json", 24)
```

3、自定义下沉器

```
package com.lagou.streamsink;
import org.apache.flink.configuration.Configuration;
import org.apache.flink.streaming.api.functions.sink.RichSinkFunction;
import org.apache.kudu.Schema;
import org.apache.kudu.Type;
import org.apache.kudu.client.*;
import org.apache.log4j.Logger;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.util.Map;

public class MySinkToKudu extends RichSinkFunction<Map<String, Object>>{

    private final static Logger logger = Logger.getLogger(MySinkToKudu.class);

    private KuduClient client;
    private KuduTable table;

    private String kuduMaster;
    private String tableName;
    private Schema schema;
    private KuduSession kuduSession;
    private ByteArrayOutputStream out;
    private ObjectOutputStream os;


    public MySinkToKudu(String kuduMaster, String tableName) {
        this.kuduMaster = kuduMaster;
        this.tableName = tableName;
    }
```

```
    @Override
    public void open(Configuration parameters) throws Exception {
        out = new ByteArrayOutputStream();
        os = new ObjectOutputStream(out);
        client = new KuduClient.KuduClientBuilder(kuduMaster).build();
        table = client.openTable(tableName);
        schema = table.getSchema();
        kuduSession = client.newSession();
        kuduSession.setFlushMode(SessionConfiguration.FlushMode.AUTO_FLUSH_BACKGROUND);
    }


    public void invoke(Map<String, Object> map) {
        if (map == null) {
            return;
        }
        try {
            int columnCount = schema.getColumnCount();
            Insert insert = table.newInsert();
            PartialRow row = insert.getRow();
            for (int i = 0; i < columnCount; i++) {
                Object value = map.get(schema.getColumnByIndex(i).getName());
                insertData(row, schema.getColumnByIndex(i).getType(),
schema.getColumnByIndex(i).getName(), value);
            }

            OperationResponse response = kuduSession.apply(insert);
            if (response != null) {
                logger.error(response.getRowError().toString());
            }
        } catch (Exception e) {
            logger.error(e);
        }
    }

    @Override
    public void close() throws Exception {
        try {
            kuduSession.close();
            client.close();
            os.close();
            out.close();
        } catch (Exception e) {
            logger.error(e);
        }
    }

    // 插入数据
    private void insertData(PartialRow row, Type type, String columnName, Object value)
throws IOException {

        try {
            switch (type) {
```

```
                case STRING:
                    row.addString(columnName, value.toString());
                    return;
                case INT32:
                    row.addInt(columnName, Integer.valueOf(value.toString()));
                    return;
                case INT64:
                    row.addLong(columnName, Long.valueOf(value.toString()));
                    return;
                case DOUBLE:
                    row.addDouble(columnName, Double.valueOf(value.toString()));
                    return;
                case BOOL:
                    row.addBoolean(columnName, (Boolean) value);
                    return;
// case INT8:
// row.addByte(columnName, (byte) value);
// return;
// case INT16:
// row.addShort(columnName, (short) value);
// return;
                case BINARY:
                    os.writeObject(value);
                    row.addBinary(columnName, out.toByteArray());
                    return;
                case FLOAT:
                    row.addFloat(columnName, Float.valueOf(String.valueOf(value)));
                    return;
                default:
                    throw new UnsupportedOperationException("Unknown type " + type);
            }
        } catch (Exception e) {
            logger.error("数据插入异常", e);
        }
    }
}
```

4、测试：

```
package com.lagou.streamsink;

import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.streaming.api.datastream.DataStreamSource;
import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;

import java.util.HashMap;
import java.util.Map;

public class SinkTest {
```

```
    public static void main(String []args) throws Exception {

        // 初始化 flink 执行环境
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        // 生成数据源
        DataStreamSource<UserInfo> dataSource = env.fromElements(
                new UserInfo("001", "Jack", 18),
                new UserInfo("002", "Rose", 20),
                new UserInfo("003", "Cris", 22),
                new UserInfo("004", "Lily", 19),
                new UserInfo("005", "Lucy", 21),
                new UserInfo("006", "Json", 24));

        // 转换数据 map
        SingleOutputStreamOperator<Map<String, Object>> mapSource = dataSource.map(new
MapFunction<UserInfo, Map<String, Object>>() {


            public Map<String, Object> map(UserInfo value) throws Exception {
                Map<String, Object> map = new HashMap<String, Object>();
                map.put("id", value.getId());
                map.put("name", value.getName());
                map.put("age", value.getAge());
                return map;
            }
        });

        // sink 到 kudu
        String kuduMaster = "hdp-2";
        String tableInfo = "user";
        mapSource.addSink(new MySinkToKudu(kuduMaster, tableInfo));

        env.execute("sink-test");
    }
}
```

实体类：

```
package com.lagou.streamsink;

public class UserInfo {
    private String id;
    private String name;
    private int age;

    public UserInfo(String id, String name, int age) {
        this.id = id;
        this.name = name;
        this.age = age;
    }
```

```
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "UserInfo{" +
                "id='" + id + '\'' +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

第六部分 kudu表设计(扩展)

Tablet是kudu表的水平分区，类似于google Bigtable的tablet，或者HBase的region。每个tablet存储着一定连续range的数据（key），且tablet两两间的range不会重叠。一张表的所有tablet包含了这张表的所有key空间。

Tablet由RowSet组成，RowSet由一组rows组成（n条数据、n行数据）。RowSet是不相交的，即不同的RowSet间的row不会交叉，因此一条给定的数据，只会存在于一个RowSet中。虽然Rowset是不相交的，但是两两间的key空间是可以相交的（key的range）。

Handling Insertions

一个RowSet存储在内存中，它被称为MemRowSet，一个tablet中只有一个MemRowSet。MemRowSet是一个in-memory的B-Tree树，且按照表的主键排序。所有的insert直接写入进MemRowSet。受益于MVCC（Multi-Version Concurrency Control 多版本并发控制，下文中会讲述），一旦数据写入到MemRowSet，后续的reader能立马查询到。

注意：不同于BigTable，Kudu只有插入和插入与flush前的mutation才会被记录到MemRowSet。mutation例如基于磁盘数据的update、deletion，下文会有介绍。

任何一条数据都以entry的形式精确的存在于一个MemRowSet中，entry由一个特殊的header和实际的row data内容组成。由于MemRowSet只存于内存中，最终会被写满，然后Flush到磁盘里（一个或者多个DiskRowSet中）。（下文会详细介绍）

MVCC overview

Kudu为了提供一些有用的特性，使用多版本并发控制：

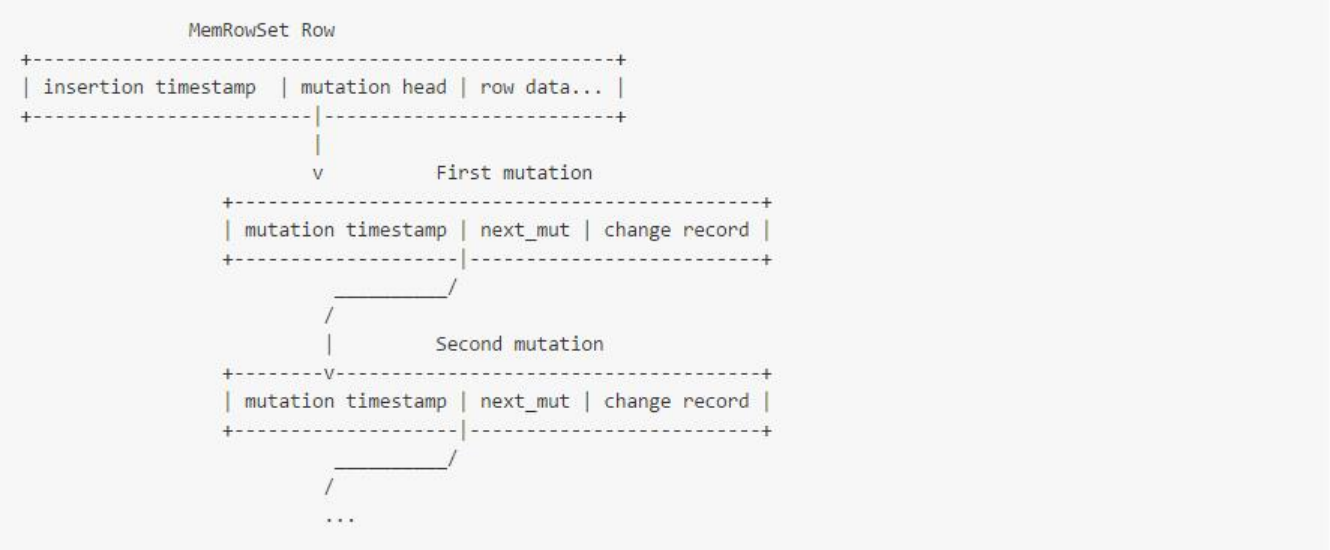
Snapshot scanner：快照查询，当创建了一个查询，系统会操作tablet指定时间的快照（point-in-time）。在这个查询过程中的任何针对这个tablet的update都会被忽略。另外，指定时间的快照（point-in-time）可以被存储并在其他的查询中重复使用，例如，一个应用对一组连续的数据进行多次交互执行分析查询。 Time-travel scanners：历史快照查询，与上边的快照查询一样。用户可以指定历史的一个时间点创建一个查询，MVCC可以保证历史快照的一致性。这个功能可以被用来在某个时间点上的一致性备份。 Change-history queries：历史变更查询，给定两个MVCC快照，用户可以查询这两个快照间任务数据。这个功能可以用来做增量备份，跨集群同步，或者离线审计分析。 Multi-row atomic updates within a tablet：tablet内多行数据的原子更新，在一个tablet里，一个操作（mutation）可以修改多行数据，而且在一条数据里的原子操作里是可见的。（应该是针对column的原子操作） 为了提供MVCC功能，每个操作（mutation）会带有一个时间戳（timestamp）。Timestamp是由TS-wide Clock实例提供的，tablet的MvccManager能保证在这个tablet中timestamp是唯一的不重复的。MvccManager决定了数据提交的timestamp，从而这个时间点后的查询都可以获取到刚刚提交的数据。查询在被创建的时候，scanner提取了一个MvccManager时间状态的快照，所有对于这个scanner可见的数据都会跟这个MvccSnapshot比较，从而决定到底是哪个insertion、update或者detete操作后的数据可见。

每个tablet的Timestamp都是单调递增的。我们使用HybridTime技术来创建时间戳，它能保证节点之间的时间戳一致。

为了支持快照和历史快照功能，多个版本的数据必须被存储。为了防止空间无限扩展，用户可以配置一个保留时间，并将这个时间之前的记录GC（这个功能可以防止每次查询都从最原始版本开始读取）。

MVCC Mutations in MemRowSet

为了在MemRowSet中支持MVCC功能，每行插入的数据都会带着时间戳。而且，row会有一个指针，它指向紧随其后的mutations列表，每个mutation都有带有timestamp：



在传统的关系型数据库术语里，这个有序的mutations列表可以被称作“RODO log”。

任何reader需要访问MemRowSet的row中的mutations，才能得到正确的快照。逻辑如下：

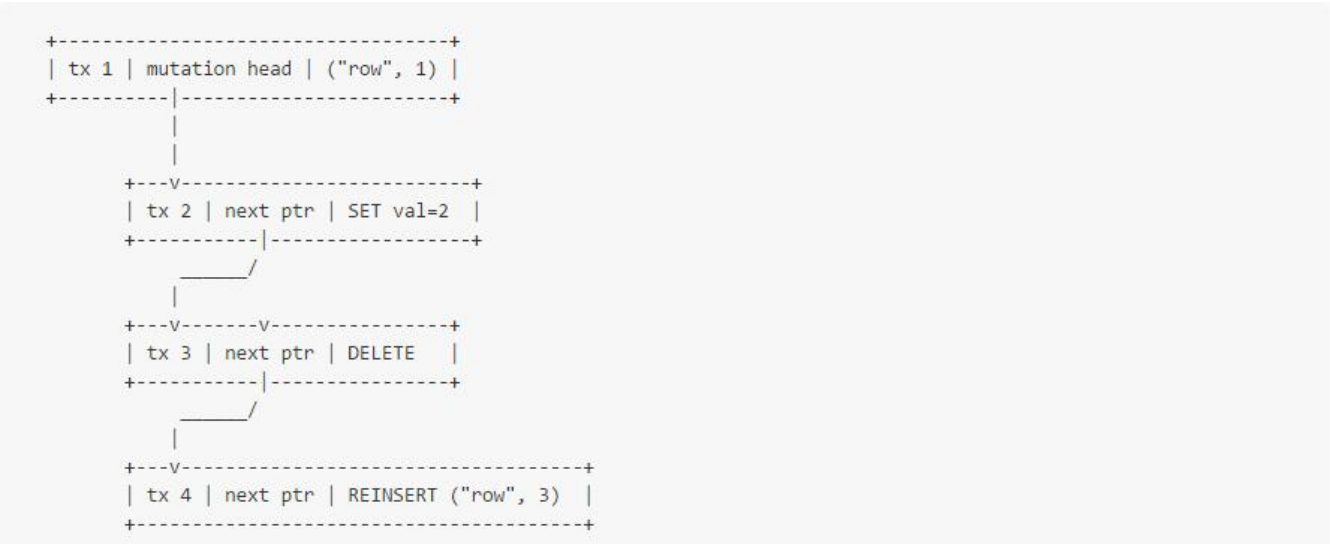
如果这行数据插入时的timestamp，不在scanner 的MVCC snapshot里（即scanner快照指定的timestamp小于数据插入的时间戳，数据还没创建），忽略该行。 如上如果不满足，将这行数据放入output缓存里。 循环list里的mutation：

1. 如果mutation的timestamp在MVCC snapshot里，在内存的缓存中执行这个更新。如果不在，则跳过此mutation。
2. 如果mutation是一个DELETE操作，则在buffer中标记为已经被删除了，并清空之前加载缓存里的数据。

注意，mutation可以是如下的任何一种：

UPDATE：更新value，一行数据里的一列或者多列 DELETE： 删除一行数据 REINSERT：重新插入一行数据（这种情况只在之前有一个DELETE mutation且数据在MemRowSet里时发生。） 举个真实例子，表结构(key STRING, val UINT32)，经过如下操作：

INSERT INTO t VALUES (“row”, 1); [timestamp 1] UPDATE t SET val = 2 WHERE key = “row”; [timestamp 2] DELETE FROM t WHERE key = “row”; [timestamp 3] INSERT INTO t VALUES (“row”, 3); [timestamp 4] 在MemRowSet中，会有如下结构：



注意，当更新过于频繁时，会有如下的影响：

readers需要追踪linked list指针，导致生成很多CPU cache任务 更新需要追加到linked list的末尾，导致每次更新的时间复杂度是O(n)。 考虑到如上低效率的操作，我们给出如下假设：

Kudu适用于相对低频率更新的场景，即假设数据不会过于频繁的更新。 整个数据中，只有一小部分存于MemRowSet中：一旦MemRowSet达到一定阈值，它会被flush到disk。因此即使MemRowSet的mutation会导致性能低，也只是占用整体查询时间的一小部分。 如果如上提到的低效率影响到了实际应用，后续会有很多降低开销的优化可以去做。

MemRowSet Flushes

当MemRowSet满了，会触发Flush操作，它会持续将数据写入disk。

数据flush到disk成了CFiles文件（参见src/kudu/cfile/README）。数据里的每行都通过一个有序的rowid标识了，而且这个rowid在DiskRowSet中是密集的、不可变的、唯一的。举个例子，如果一个给定的DiskRowSet包含有5行数据，那么它们会以key上升的顺序被分配为rowid0~4。不同的DiskRowSet，会有不同的行（rows），但却可能有相同rowid。

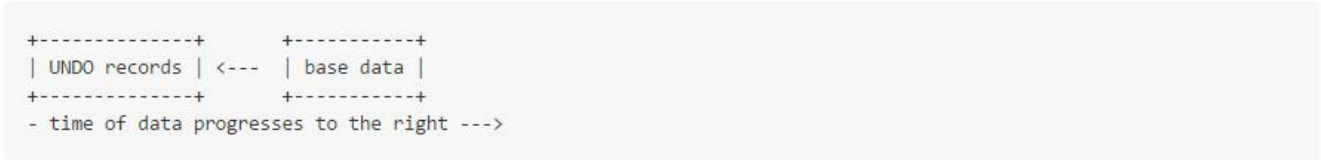
读取时，系统会使用一个索引结构，把用户可见的主键key和系统内部的rowid映射起来。上述例子中的主键是一个简单的key，它的结构嵌入在主键列的cfile里。另外，一个独立的index cfile保存了编码后的组合key，使用了提供了类似的方法。（不懂）

注意：rowid不是精确的跟每行数据的data存在一起，而是在这个cfile里根据数据有序的index的一个隐式识别。在一部分源码中，将rowid定义为 “row indexes” 或者 “ordinal indexes”。

注意：其他系统，例如C-Store把MemRowSet称为”write optimized store” (WOS)，把DiskRowSet称为”read-optimized store” (ROS)。

Historical MVCC in DiskRowSets

为了让on-disk data具备MVCC功能，每个on-disk的Rowset不仅仅包含当前版本row的data，还包含UNDO的记录，如此，可以获取这行数据的历史版本。



当用户想读取flush后最新版本的数据时，只需要获取base data。因为base data是列式存储的，这种查询性能是非常高的。如果不是读取最新数据，而是time-travel查询，就得回滚到指定历史时间的一个版本，此时就需要借助UNDO record数据。

当一个查询拿到一条数据，它处理MVCC信息的流程是：

读取base data 循环每条UNDO record：如果相关的操作timestamp还未提交，则执行回滚操作。即查询指定的快照timestamp小于mutation的timestamp，mutation还未发生。 举个例子，回顾一下之前MVCC Mutations in MemRowSet章节例子的一系列操作：

```
INSERT INTO t VALUES ("row", 1);         [timestamp 1]
UPDATE t SET val = 2 WHERE key = "row";  [timestamp 2]
DELETE FROM t WHERE key = "row";         [timestamp 3]
INSERT INTO t VALUES ("row", 3);         [timestamp 4]
```

当这条数据flush进磁盘，它将会被存成如下形式：

```
Base data:
   ("row", 3)
UNDO records (roll-back):
   Before Tx 4: DELETE
   Before Tx 3: INSERT ("row", 2")
   Before Tx 2: SET row=1
   Before Tx 1: DELETE
```

每条UNDO record是执行处理的反面。例如在UNDO record里，第一条INSERT事务会被转化成DELETE。UNDO recod旨在保留插入或者更新数据的时间戳：查询的MVCC快照指定的时间早于Tx1时，Tx1还未提交，此时将会执行DELETE操作，那么这时这条数据是不存在的。

再举两个不同查询的例子：

```
Current time scanner (all txns committed)
------------------------------------------
- Read base data
- Since tx 1-4 are committed, ignore all UNDO records
- No REDO records
Result: current row ("row", 3)


Scanner as of timestamp 1
---------------------
- Read base data. Buffer = ("row", 3)
- Rollback Tx 4:  Buffer = <deleted>
- Rollback Tx 3:  Buffer = ("row", 2)
- Rollback Tx 2:  Buffer = ("row", 1)
Result: ("row", 1)
```

每个例子都处理了正确时间的UNDO record，以产生正确的数据。

最常见的场景是查询最新的数据。此时，我们需要优化查询策略，避免处理所有的UNDO records。为了达到这个目标，我们引入文件级别的元数据，指向UNDO record的数据范围。如果查询的MVCC快照符合的所有事务都已经提交了（查询最新的数据），这组deltas就会短路（不处理UNDO record），这时查询将没有MVCC开销。

Handling mutations against on-disk files

更新或者删除已经flush到disk的数据，不会操作MemRowSet。它的处理过程是这样的:为了确定update/delete的key在哪个RowSet里，系统将巡视所有RowSet。这个处理首先使用一个区间tree，去定位一组可能含有这key的RowSet。然后，使用boom filter判断所有候选RowSet是否含有此key。如果某一些RowSet同时通过了如上两个check，系统将在这些RowSet里寻找主键对应的rowid。

一旦确定了数据所在的RowSet，mutation将拿到主键对应的rowid，然后mutation会被写入到一个称为DeltaMemStore的内存结构中。

一个DiskRowSet里就一个DeltaMemStore，DeltaMemStore是一个并行BTree，BTree的key是使用rowid和mutation的timestamp混合成的。查询时，符合条件的mutation被执行后得到快照timestamp对应数据，执行方式与新数据插入后的mutation类似（MemRowSet）。

当DeltaMemStore存入的数据很大后，同样也会执行flush到disk，落地为DeltaFile文件:

```
+-------------+      +---------+     +---------+     +-----------------+
| base data   | <--- | delta 0 | <-- | delta N | <-- | delta memstore |
+-------------+      +---------+     +---------+     +-----------------+
```

DeltaFile的信息类型与DeltaMemStore是一致的，只是被压实和序列化在密集型的磁盘里。为了把数据从base data更新成最新的数据，查询时需要执行这些DeltaFile里的mutation事务，这些DeltaFile集合称作REDO文件，而file里这些mutation称作REDO record。与存于MemRowSet里的mutation类似，当读取比base data更新版本的数据时，它们需要被一次应用（执行）。

一条数据的delta信息可能包含在多个DeltaFile文件，这种情况下，DeltaFile是有序的，后边的变更会优先于前边的变更。

注意，mutation存储结构没必要包含整行的数据。如果在一行中，仅仅只有一列数据被更新，那么mutation结构只会包含这一列的更新信息。不读取或者重写无关的列，这样更新数据操作就快而有效率。

Summary of delta file processing

总结一下，每个DiskRowSet逻辑上分三部分：

```
+---------------+       +------------+      +---------------+
| UNDO records | <---   | base data | ---> | REDO records |
+---------------+       +------------+      +---------------+
```

Base data：MemRowSet flush到DiskRowSet时的最新数据，数据是列式存储的。 UNDO records：历史数据，用来回滚到Base data之前一些历史版本数据。 REDO records：Base data之后的一些更新数据，可以用来得到最新版本的数据。 UNDO record 和REDO record存储格式是一样的，都称为DeltaFile。

Delta Compactions

当DeltaFile里的mutation堆积越来越多，读取RowSet数据效率就越来越低，最坏情况，读取最新版本数据需要遍历所有REDO record并与base data merge。换一句话说，如果数据被更新了太多次，为了得到最新版本的数据，就需要执行这么多次的mutation。

为了提高读取性能，Kudu在后台将低效率的物理布局转化成更加高效的布局，且转化后具有同样的逻辑内容。这种转化称为：delta compaction。它的目标如下：

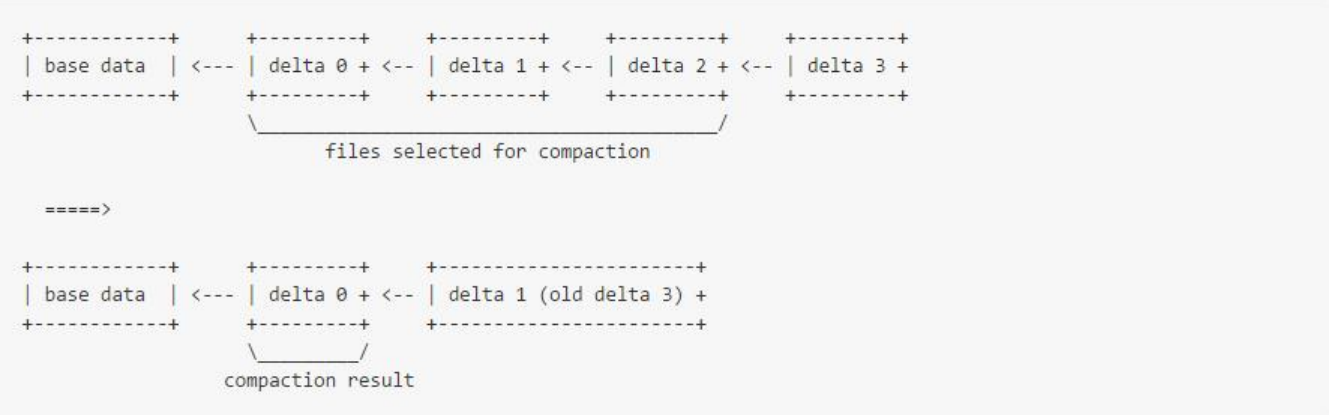
1. 减少delta files数量。RowSet flush的delta files文件越多，为了读取最新版本数据所要读取的独立的delta files就越多。这个工作不适于放在内存中（RAM），因为每次读取都会带有delta file的磁盘寻址，会遭受性能损失。
2. 将REDO records迁移成UNDO records。如上所述，一个RowSet包含了一个base data，且是按列存储的，往后一段是UNDO records，往前一段是REDO records。大部分查询都是为了获取最新版本的数据，因此我们需要最小化REDO records数量。
3. 回收old UNDO records。UNDO recods只需要保存用户设定最早时间点后的数据，这个时间之前的UNDO record都可以从磁盘中移除。

注意：BigTable的设计是timestamp绑定在data里，没有保留change信息（insert update delete）；而kudu的设计是timestamp绑定在change里，而不是data。如果历史的UNDO record被删除，那么将获取不到某行数据或者某列数据是什么时候插入或者更新的。如果用户需要这个功能，他们需要保存插入或者更新的timestamp列，就跟传统关系型数据库一样。

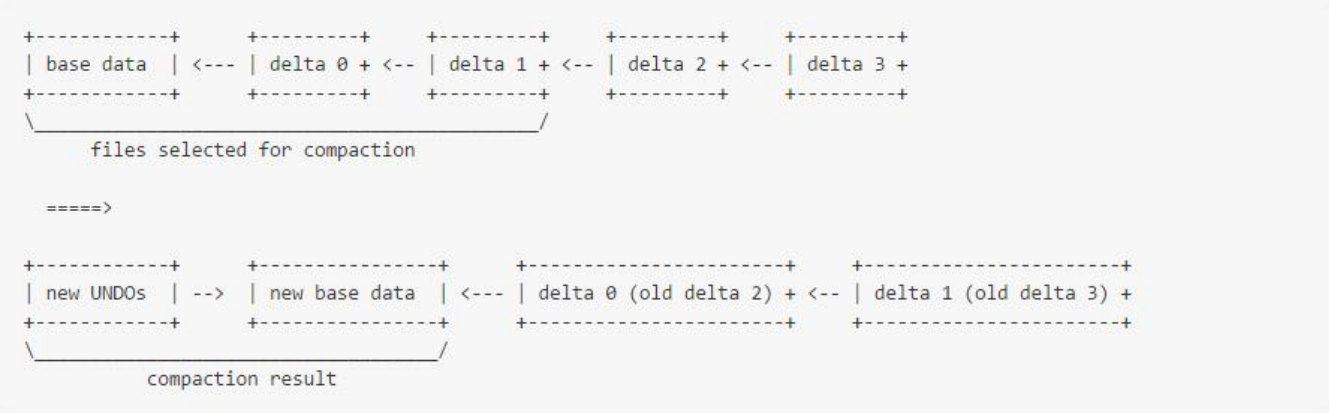
Types of Delta Compaction

delta campaction分minor和major两种。

Minor delta compactoin： Minor compaction是多个delta file的compaction，不会包含base data，compact生成的也是delta file。



Major delta compaction： Major compaction是对base data和任意多个delta file的compact。



Major compaction比minor compaction更耗性能，因为它需要读取和重写base data，并且base data比delta data大很多（因为base data存了一行数据，而delta data是对某一些column的mutation，需要注意的base data是列式存储的，delta data不是）。

Major compaction可以对DiskRowSet里的任意多个或者一个column进行compact。如果只有一列数据进行了多次重要的更新，那么compact可以只针对这一列进行读取和重写。在企业级应用中会经常遇到这种情况，例如更新订单的状态、更新用户的访问量。

两种类型的compaction都维护RowSet里的rowid。因为它们完全在后台执行，且不会带锁。compact的结果文件会采用原子swapping的方式被引入进RowSet。Swap操作结束后，compact前的那些老文件将会被删除。

Merging compactions

随着越来越多的数据写入tablet，DiskRowSet数量也会累积的越来越多。如此这般将会降低kudu性能：

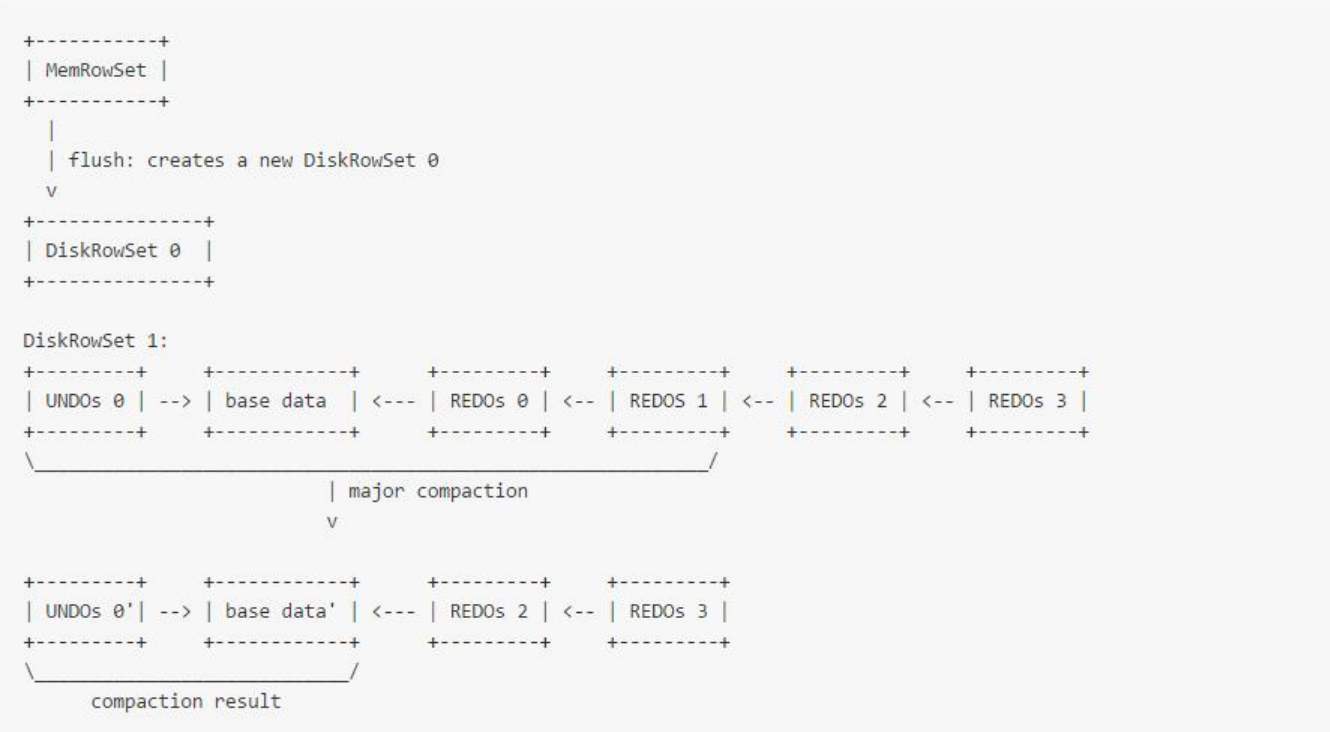
1. 随机访问（通过主键获取或者更新一条数据），这种情况下，每个RowSet只要它的key范围包含了这个主键，将各自去定位主键的位置。Boom filter可以缓解一定数量的物理寻址，但是特大的bloom filter访问会影响到CPU，并且同样会增加内存消耗。

2. 查询一定key范围数据（例如查询主键在A与B之间的数据），此时，每个RowSet，只要它的key范围与提供的范围重叠，将各自去寻址，不使用bloom filter。专门的索引结构可能会有帮助，但是同样会消耗内存。
3. 排序查询，如果用户要求查询的结果与主键有相同顺序，那么查询结果集必须经过一个merge过程。Merge的消耗通常与输入的数据量成对数级增长，即随着数据量的增大，merge将越耗性能。

如上所述，我们应该merge RowSet以减少RowSet的数量：

与如上提到的Delta Compaction不同，请注意，merging Compaction不会保持rowid一样。这使得处理并发的mutation错综复杂。这个过程在compaction.txt文件中有比较详细的描述。

Overall picture



Comparison to BigTable approach

与BigTable不同的设计方式点如下：

1. kudu中，一个给定的key只会存在于一个tablet的RowSet里。 在BigTable里，一个key可以存在多个不同的SSTable里。BigTable的一整个Tablet类似于kudu的RowSet：读取一条数据需要merge所有SSTable里找到的数据（根据key），类似于Kudu，读取一条数据需要merge base data和所有DeltaFile数据。 Kudu的优势是，读取一条数据或者执行非排序查询，不需要merge。例如，聚合一定范围内的key可以独立的查询每个RowSet(甚至可以并行的)，然后执行求和，因为key的顺序是不重要的，显然查询的效率更高。 Kudu的劣势是，不像BigTable，insert和mutation是不同的操作：insert写入数据至MemRowSet，而mutation（delete、update）写入存在这条数据的RowSet的DeltaMemStore里。性能影响有一下几点： a）写入时必须确定这是一条新数据。这会产生一个bloom filter查询所有RowSet。如果布隆过滤器得到一个可能的match（即计算出可能在一个RowSet里），接着为了确定是否是insert还是update，一个寻址就必须被执行。 假设，只要RowSet足够小，bloom filter的结果就会足够精确，那么大部分插入将不需要物理磁盘寻址。另外，如果插入的key是有序的，例如timeseries+“_”+xxx，由于频繁使用，key所在的block可能会被保存在数据块缓存中。 b）Update时，需要确定key在哪个RowSet。与上雷同，需要执行bloom filter。

   这有点类似于关系型数据库RDBMS，当插入一条主键存在的数据时会报错，且不会更新这条数据。类似的，更新一条数据时，如果这条数据不存在也会报错。BigTable的语法却不是这样。

2. Mutation操作磁盘数据，是通过rowid的，而不是实际意义上的key。 BigTable中，同一个主键数据是可以存在多个SSTable里的，为了让mutation和磁盘的存的key组合在一起，BigTable需要基于rowkey执行merge。Rowkey可以是任意长度的字符串，因此对比rowkey是非常耗性能的。另外，在一个查询中，即使key列没有被使用（例如聚合计算），它们也要被读取出来，这导致了额外的IO。复合主键在BigTable应用中很常见，主键的大小可能比你关注的列大一个数量级，特别是查询的列被压缩的情况下。 相比之下，kudu的mutation是与rowid绑定的。所以merge会更加高效，通过维护计数器的方式：给定下一个需要保存的mutation，我们可以简单的相减，就可以得到从base data到当前版本有多少个mutation。或者，直接寻址可以用来高效的获取最新版本的数据。 另外，如果在查询中没有指定key，那执行计划就不会查阅key，除了需要确定key边界情况。 举例：

   > SELECT SUM(cpu_usage) FROM timeseries WHERE machine = 'foo.cloudera.com' AND unix_time BETWEEN 1349658729 AND 1352250720; ... given a composite primary key (host, unix_time)

   如上表的主键是(host,unitx_time)，在kudu里的执行伪代码如下：sum = 0 foreach RowSet: start_rowid = rowset.lookup_key(1349658729) end_rowid = rowset.lookup_key(1352250720) iter = rowset.new_iterator(“cpu_usage”) iter.seek(start_rowid) remaining = end_rowid - start_rowid while remaining > 0: block = iter.fetch_upto(remaining) sum += sum(block)。 获取block也非常的高效，因为mutation直接指向了block的索引地址。

3. timgstamp不是数据模型里的一部分。 BigTable-like的系统中，每个cell的timstamp都是暴露给用户的，本质上组成了这个cell的一个符合主键。意味着，这种方式可以高效的直接访问指定版本的cell，且它存储了一个cell的整个时间序列的所有版本。而Kudu却不高效（需要执行多个mutation），它的timestamp是从MVCC实现而来的，它不是主键的另外一个描述。

   作为替代，kudu可以使用原生的复合主键来满足时间序列场景，例如主键（host，unix_time）。