基于 Kudu 的分布式数据库部署和优化

5140309254 林顺达

2017 年 5 月 9 日

一、　背景介绍

Acemap 作为学术搜索引擎，平时需要对数据库进行大量的查询操作。当数据量较为庞大时，进行复杂的查询需要花费较多的时间，所以希望能对数据库的查询速度进行优化。目前网站使用的数据库为 MySQL，我们小组尝试改用分布式数据库来提升查询速度，我负责 Kudu 这个方向。

二、　Kudu 概况

Kudu 是 Cloudera 开源的新型列式存储系统，是 Apache Hadoop 生态圈的新成员之一（incubating），专门为了对快速变化的数据进行快速的分析，填补了以往 Hadoop 存储层的空缺。

传统的实时/离线计算架构几乎都是分离的，用户需要把实时的数据从一个实时队列系统导入到在线存储，再导出到离线存储系统。有时候离线和实时的数据又存在 join 或并用的可能性。

传统系统几大痛点：

1、应用系统需要在实时、离线系统之间把数据倒来倒去，写很复杂的 code

2、系统庞杂，各种备份，安全策略，监控系统

3、系统从实时系统中流出到离线系统才好做 OLAP 分析，这层转换存在延迟

4、现实上来说，系统总是会存在落后的数据、对过去数据的修改，删除。然而当过去的数据已经被归档，这些操作需要昂贵的重写以及 partiion 交换或者各种人工干预

Cloudera 很早就意识到这个问题，在 2012 年就开始计划开发 Kudu 这个存储系统，终于在 2015 年发布并开源出来。Kudu 是对 HDFS 和 HBase 功能上的补充，能提供快速的分析和实时计算能力，并且充分利用 CPU 和 I/O 资源，支持数据原地修改，支持简单的、可扩展的数据模型。Kudu 弥补了高速顺序吞吐系统和低延迟随机访问系统之间的 gap。它同时提供行级别的插入，更新，删除。也提供类似 Parquet 的批量 scan，列读取等。

三、　Kudu 架构与设计

1.基本框架

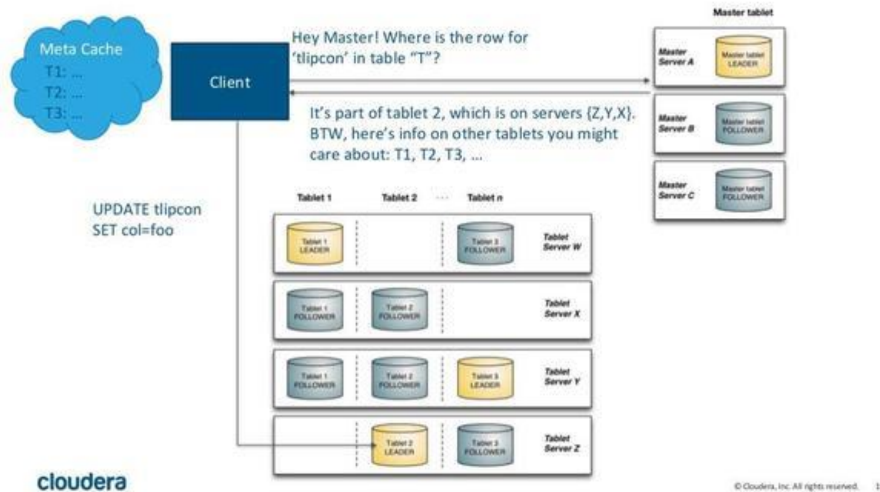
Kudu 是用于存储结构化（structured）的表（Table）。表有预定义的带类型的列（Columns），每张表有一个主键（primary key）。主键带有唯一性（uniqueness）限制，可作为索引用来支持快速的 random access。

类似于 BigTable，Kudu 的表是由很多数据子集构成的，表被水平拆分成多个 Tablets. Kudu 用以每个 tablet 为一个单元来实现数据的 durability。Tablet 有多个副本，同时在多个节点上进行持久化。

Kudu 有两种类型的组件，Master Server 和 Tablet Server。Master 负责管理元数据。这些元数据包括 talbet 的基本信息，位置信息。Master 还作为负载均衡服务器，监听 Tablet Server 的健康状态。对于副本数过低的 Tablet，Master 会在起 replication 任务来提高其副本数。Master 的所有信息都在内存中 cache，因此速度非常快。每次查询都在百毫秒级别。Kudu 支持多个 Master，不过只有一个 active Master，其余只是作为灾备，不提供服务。

Tablet Server 上存了 10~100 个 Tablets，每个 Tablet 有 3（或 5）个副本存放在不同的 Tablet Server 上，每个 Tablet 同时只有一个 leader 副本，这个副本对用户提供修改操作，然后将修改结果同步给 follower。Follower 只提供读服务，不提供修改服务。副本之间使用 raft 协议来实现 High Availability，当 leader 所在的节点发生故障时，followers 会重新选举 leader。根据官方的数据，其 MTTR 约为 5 秒，对 client 端几乎没有影响。Raft 协议的另一

个作用是实现 Consistency。Client 对 leader 的修改操作，需要同步到 N/2+1 个节点上，该操作才算成功。



Kudu 采用了类似 log-structured 存储系统的方式，增删改操作都放在内存中的 buffer，然后才 merge 到持久化的列式存储中。Kudu 还是用了 WALs 来对内存中的 buffer 进行灾备。

2.列式存储

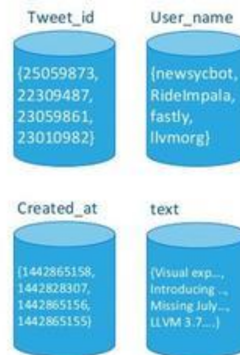
持久化的列式存储存储，与 HBase 完全不同，而是使用了类似 Parquet 的方式，同一个列在磁盘上是作为一个连续的块进行存放的。例如，图中左边是 twitter 保存推文的一张表，而图中的右边表示了表在磁盘中的的存储方式，也就是将同一个列放在一起存放。这样做的第一个好处是，对于一些聚合和 join 语句，我们可以尽可能地减少磁盘的访问。例如，我们要用户名为 newsycbot 的推文数量，使用查询语句：

SELECT COUNT(*) FROM tweets WHERE user_name = ‘newsycbot’；

Columnar Storage

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



User_name
{newsycbot, RideImpala, fastly, llvmorg}

text
{Visual exp..., Introducing ..., Missing July..., LLVM 3.7...}

cloudera

我们只需要查询 User_name 这个 block 即可。同一个列的数据是集中的，而且是相同格式的，Kudu 可以对数据进行编码，例如字典编码，行长编码，bitshuffle 等。通过这种方式可以很大的减少数据在磁盘上的大小，提高吞吐率。除此之外，用户可以选择使用通用的压缩格式对数据进行压缩，如 LZ4, gzip，或 bzip2。这是可选的，用户可以根据业务场景，在数据大小和 CPU 效率上进行权衡。这一部分的实现上，Kudu 很大部分借鉴了 Parquet 的代码。

HBase 支持 snappy 存储，然而因为它的 LSM 的数据存储方式，使得它很难对数据进行

特殊编码，这也是 Kudu 声称具有很快的 scan 速度的一个很重要的原因。不过，因为列式编码后的数据很难再进行修改，因此当这写数据写入磁盘后，是不可变的，这部分数据称之为 base 数据。Kudu 用 MVCC（多版本并发控制）来实现数据的删改功能。更新、删除操作需要记录到特殊的数据结构里，保存在内存中的 DeltaMemStore 或磁盘上的 DeltaFIle 里面。DeltaMemStore 是 B-Tree 实现的，因此速度快，而且可修改。磁盘上的 DeltaFIle 是二进制的列式的块，和 base 数据一样都是不可修改的。因此当数据频繁删改的时候，磁盘上会有大量的 DeltaFiles 文件，Kudu 借鉴了 Hbase 的方式，会定期对这些文件进行合并。

3.对外接口

Kudu 提供 C++和 JAVA API，可以进行单条或批量的数据读写，schema 的创建修改。除此之外，Kudu 还将与 hadoop 生态圈的其它工具进行整合。目前，kudu beta 版本对 Impala 支持较为完善，支持用 Impala 进行创建表、删改数据等大部分操作。Kudu 还实现了 KuduTableInputFormat 和 KuduTableOutputFormat，从而支持 Mapreduce 的读写操作。同时支持数据的 locality。

四、 Kudu 配置

Kudu 集群的配置用到了服务器上的三台机器，操作系统都为 Ubuntu 16.04：

Slave1--192.168.0.135(tserver)

Slave2--192.168.0.134(master、tserver)

Slave3—192.168.0.100(tserver)

（1）将 cloudera.list 文件保存到/etc/apt/sources.list.d/目录下

```
# Packages for Cloudera's Distribution for Hadoop, Version 5, on Ubuntu 16.04 amd64
deb [arch=amd64] http://archive.cloudera.com/kudu/ubuntu/xenial/amd64/kudu xenial-kudu5 contrib
deb-src http://archive.cloudera.com/kudu/ubuntu/xenial/amd64/kudu xenial-kudu5 contrib
```

（2）更新源: apt-get update

（3）安装 Kudu

```
sudo apt-get install kudu                     # Base Kudu files
sudo apt-get install kudu-master              # Service scripts for managing kudu-master
sudo apt-get install kudu-tserver             # Service scripts for managing kudu-tserver
sudo apt-get install libkuduclient0           # Kudu C++ client shared library
sudo apt-get install libkuduclient-dev        # Kudu C++ client SDK
```

（4）在 /etc/kudu/conf 和 /etc/default 配置路径和主节点以及子节点。 如下：

```
[root@host3 conf]# more master.gflagfile
 # Do not modify these two lines. If you wish to change these variables,
 # modify them in /etc/default/kudu-master.
 --fromenv=rpc_bind_addresses
 --fromenv=log_dir

 # --fs_wal_dir=/var/lib/kudu/master
 # --fs_data_dirs=/var/lib/kudu/master
 --fs_wal_dir=/opt/kudu/master
 --fs_data_dirs=/opt/kudu/master
```

```
[root@host3 conf]# pwd
 /etc/kudu/conf
 [root@host3 conf]# more tserver.gflagfile
 # 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
 --fs_wal_dir=/opt/kudu/tserver
 --fs_data_dirs=/opt/kudu/tserver
 --tserver_master_addrs=host1:7051      #这里是主节点 host1，所有节点设置好之
后启动会根据这里的主节点创建 kudu 集群
```

上面是/etc/kudu/conf 中的配置文件，下面是/etc/default 中的文件（同样是修改两个文件），因为配置的是 3 台的集群，默认的备份是 3，所以不需要配置备份数。如果不是的话，需要自行配置备份数。

```
[root@host3 default]# ls
 kudu-master   kudu-tserver   nss   useradd
 [root@host3 default]# pwd
 /etc/default
 [root@host3 default]# more kudu-master
 export FLAGS_log_dir=/var/log/kudu
 export FLAGS_rpc_bind_addresses=host1:7051          #这里是主节点
 [root@host3 default]# more kudu-tserver
 export FLAGS_log_dir=/var/log/kudu
 export FLAGS_rpc_bind_addresses=host3:7050           #这里是本机
```

（5）启动 master 和 tserver 的命令分别为：

sudo service kudu-master start

sudo service kudu-tserver start

在 Slave2 这台服务器上启动 master，在 Slave1、Slave2、Slave3 这三台服务器上都启动 tserver

（6）查看集群状态：kudu cluster ksck 192.168.0.134

五、数据持久化

现有的数据都存储在 MySQL 中，需要将数据从 MySQL 迁移到 Kudu。迁移分两步：

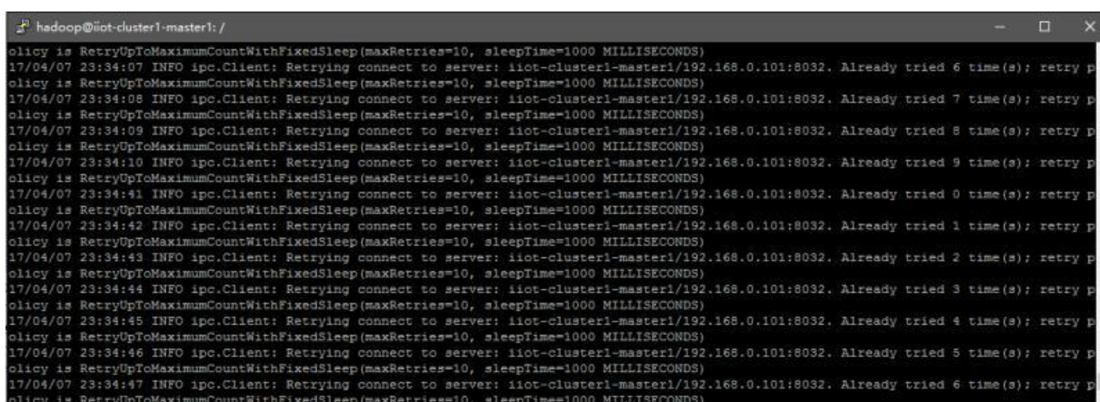
（1）数据从 MySQL 迁移到 HDFS

这个过程使用到了 Sqoop 将 MySQL 中的 Table 迁移成 HDFS 中的 parquet。

使用的命令如下：

```
Sqoop import –connect jdbc:mysql://202.120.36.137:6033/mag-new-160205 –username=data –password=data –table AuthorFieldCount –m 1 –target-dir /user/hadoop/AuthorFieldCount –as-parquetfile
```

期间碰到过问题，连接 resourcemanager 时出现错误：



解决方法是重新开启 yarn，命令为： start-yarn.sh

迁移成功后通过网页 202.120.36.28:50070 查看已迁移的数据：

Browse Directory

/user/hadoop Go!

Permission	Owner	Group	Size	Last Modified	Replication	Block Size	Name
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月10日 20:38:43	0	0 B	.sparkStaging
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月21日 21:03:45	0	0 B	.temp
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月17日 10:56:58	0	0 B	AuthorFieldCount
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月8日 22:49:01	0	0 B	Authors
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月21日 21:03:45	0	0 B	FOSReferencesCount
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月8日 23:07:09	0	0 B	FieldsOfStudy
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月17日 9:56:09	0	0 B	Linsd
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月24日 12:07:44	0	0 B	Liuyuezhou
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月4日 15:54:03	0	0 B	PaperAuthorAffiliations
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月8日 22:05:45	0	0 B	PaperKeywords
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月8日 23:08:57	0	0 B	PaperRecommenderList
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月4日 12:09:37	0	0 B	PaperReferences
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月8日 21:23:47	0	0 B	PaperSciReferencesCount
drwxr-xr-x	hadoop	supergroup	0 B	2017年4月1日 23:51:59	0	0 B	Venues_Network

（2）数据从 HDFS 迁移到 Kudu

数据迁移到 Kudu 需要先在 Kudu 中建表，然后插入数据。Kudu 本身只提供了增删查改的接口，创建表需要依赖别的 api。Kudu 提供了丰富的 api，像 C++、Java、Python、Spark、Impala 等等。目前网上 Kudu 的例子多是使用 Impala 与 Kudu 配合使用，但由于 Impala 不支持 Ubuntu 系统，所以选择使用 Spark 对 Kudu 进行操作。

Spark 是 UC Berkeley AMP lab 所开源的类 Hadoop MapReduce 的通用的并行计算框架，Spark 基于 map reduce 算法实现的分布式计算，拥有 Hadoop MapReduce 所具有的优点；但不同于 MapReduce 的是 Job 中间输出和结果可以保存在内存中，从而不再需要读写 HDFS，因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 map reduce 的算法。

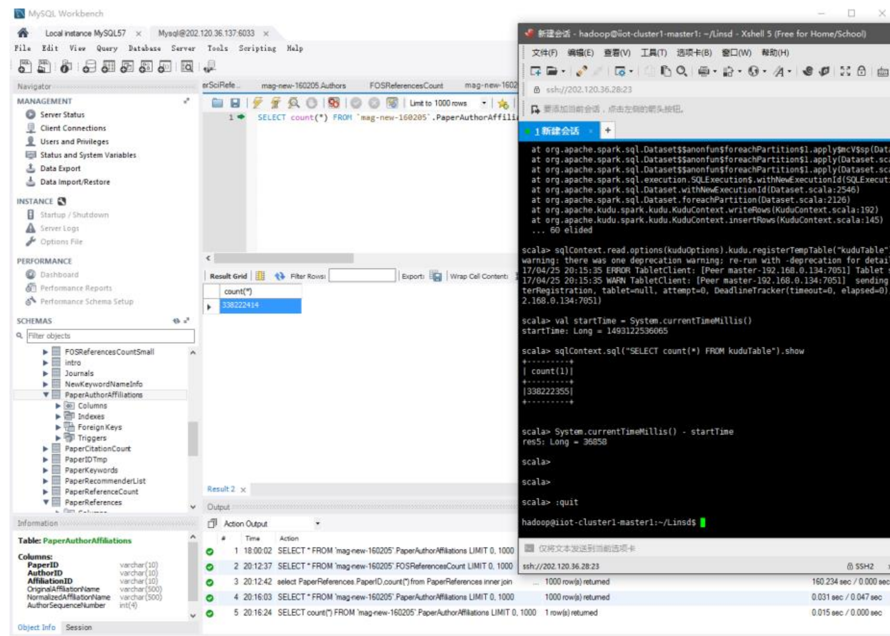
这里所使用的 Spark 版本为 2.0.1。为了方便，我使用的是 spark-shell（脚本运行模式），将 scala 程序保存在文件中，可以通过以下命令一次运行该文件中的程序代码：

```
spark-shell --packages org.apache.kudu:kudu-spark2_2.11:1.3.0  --executor-memory 10g  --num-executors 20 --executor-cores 3 --driver-memory 1g --conf spark.default.parallelism=100 --master spark://192.168.0.101:7077<createTable_PaperAuthorAffiliations
```

createTable_PaperAuthorAffiliations 中的代码用于创建 PaperAuthorAffiliations 这个表，并进行数据迁移，代码如下：

```
import org.apache.hadoop.conf.Configuration
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.sql.types._
import org.apache.spark.sql.{DataFrame, Row, SQLContext}
import scala.collection.JavaConverters._
import org.apache.spark.sql.functions._
import org.apache.kudu.spark.kudu._
import org.apache.kudu.client._



val sparkConf = new SparkConf().setAppName("create table on Kudu")

val sc = new SparkContext(sparkConf)
val sqlContext = new SQLContext(sc)

val kuduMasters = "192.168.0.134"
val kuduContext = new KuduContext(kuduMasters)
import sqlContext.implicits._
var kuduTableName = "PaperAuthorAffiliations"
val kuduOptions: Map[String, String] = Map("kudu.table"  -> kuduTableName,"kudu.master" -> kuduMasters)

val kuduTableSchema = StructType(StructField("PaperID", StringType , true) ::StructField("AuthorID" , StringType, true ) ::StructField("AffiliationID", StringType,true) :: StructField("OriginalAffiliationName",StringType,true)::StructField("NormalizedAffiliationName",StringType,true)::StructField("AuthorSequenceNumber",IntegerType,true)::Nil)
val kuduPrimaryKey = Seq("PaperID","AuthorID")
val kuduTableOptions = new CreateTableOptions()
kuduTableOptions.addHashPartitions(List("AuthorID").asJava,3)
kuduContext.createTable(kuduTableName, kuduTableSchema, kuduPrimaryKey, kuduTableOptions)
val df1 = sqlContext.read.parquet("PaperAuthorAffiliations/e3eec483-0e0d-4c17-bd3e-a630c807dc55.parquet")
kuduContext.insertRows(df1, kuduTableName)
sqlContext.read.options(kuduOptions).kudu.registerTempTable("kuduTable")

val startTime = System.currentTimeMillis()
sqlContext.sql("SELECT count(*) FROM kuduTable").show
```

在创建 PaperReferences 和 PaperAuthorAffiliations 这两个表时出现了问题，插入数据时报了错。一个报错是显示 AffiliationID 这个字段不能为 null，这个是因为创建表时设置联合主键把 AffiliationID 也设置为了主键，但在 kudu 中主键的值不能为 null。删除表后不把 AffiliationID 设置为主键重新创建了表，问题解决，但因此导致数据不全，缺失了 59 行。



另一个报错是出现了未知的字段”MagProvide”，但 Table 里实际上并没有这个 column。后来发现 MySQL 里的表 PaperReferences 中的 UpdateTime 字段其实就是 MagProvide，重新设计 Schema 后数据迁移成功，不过迁移后数据少了 845627 行。总结后我认为上述这两个报错都是由于 MySQL 中的数据存在“脏”数据造成的，这也反映出 Kudu 对于数据格式的要求较高。

六、 SQL 查询测试

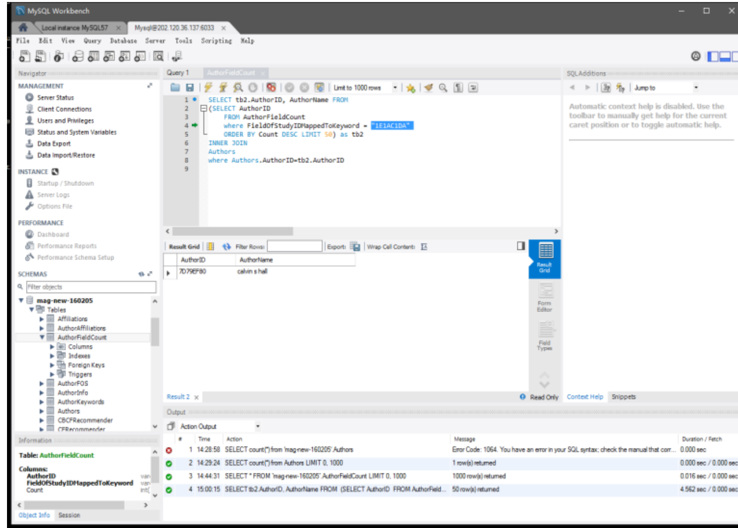
数据迁移到 **Kudu** 以后就可以开始进行 **SQL** 语句的测试了。**Kudu** 对 **SQL** 语句的支持依赖其它的工具如 **Impala**、**Spark**，这里还是使用 **Spark** 进行 **SQL** 语句的查询。

（1）找出被某个领域引用最多的 50 个作者

```
SELECT tb2.AuthorID, AuthorName FROM
(SELECT AuthorID
    FROM AuthorFieldCount
    where FieldOfStudyIDMappedToKeyword = "1E1AC1DA"
    ORDER BY Count DESC LIMIT 50) as tb2
INNER JOIN
Authors
where Authors.AuthorID=tb2.AuthorID
```

用到了两个表，AuthorFieldCount（81643611 条记录）和 Authors（114796077 条记录）。

MySQL 运行结果，耗时 4.562s：

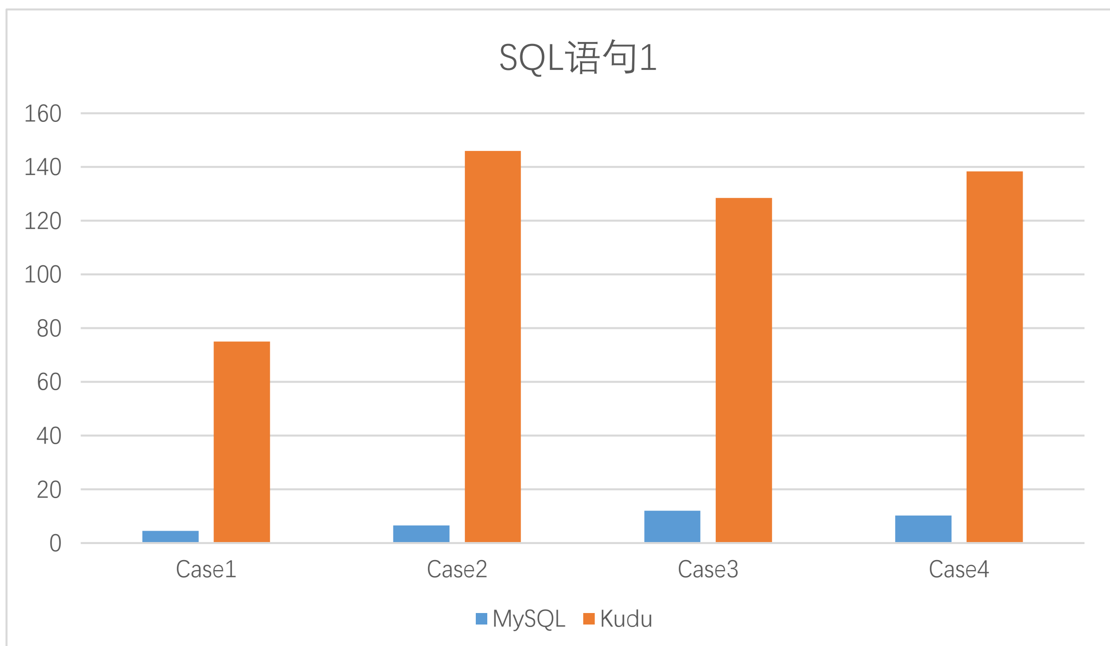


Kudu 测试结果，耗时 75s：

```
scala> val startTime = System.currentTimeMillis()
startTime: Long = 1492498697670

scala>

scala> sqlContext.sql("SELECT tb2.AuthorID, AuthorName FROM (SELECT AuthorID FROM AuthorFieldCount where FieldOfStudyIDMappedToKeyword =  \"1E1AC1DA\"  ORDER BY Count DESC LIMIT 50) as tb2 INNER JOIN Authors where Authors.AuthorID=tb2.Au
thorID").show
+--------+------------------+
|AuthorID|        AuthorName|
+--------+------------------+
|8638BA8A|  gregory d webster|
|863EE2D9|   roy f baumeister|
|7D957773|     daniel n jones|
|758E2606|        sara konrath|
|77B75270|       nicole m cain|
|77C9776F|   richard w robins|
|6CFB787F| robert gary raskin|
|7D006F07|   roy f baumeister|
|7D27A11D|   w keith campbell|
|7D79EF80|       calvin s hall|
|0A86B163|       jean m twenge|
|57082C31|    peter k jonason|
|57A43F85|     brad j bushman|
|78F6EE82|       robert raskin|
|7912AADB|  gregory d webster|
|7E4DF37C|christopher t barry|
|1156EA25| virgil zeiglerhill|
|7F735495|    joshua d foster|
|7F9B6076|   michael c ashton|
|803CB2C3|     craig a foster|
+--------+------------------+
only showing top 20 rows


scala>

scala> System.currentTimeMillis() - startTime
res3: Long = 75028
```

更换不同的参数得到如下结果：

PaperKeyword	narcissism (1E1AC1DA)	Graphicshardware (012117A5)	Machine learning (0724DFBA)	virtual reality (0B9EFF3D)
MySQL	4.562s	6.609s	12.031s	10.266s
Kudu	75s	146s	128.45s	138.399s

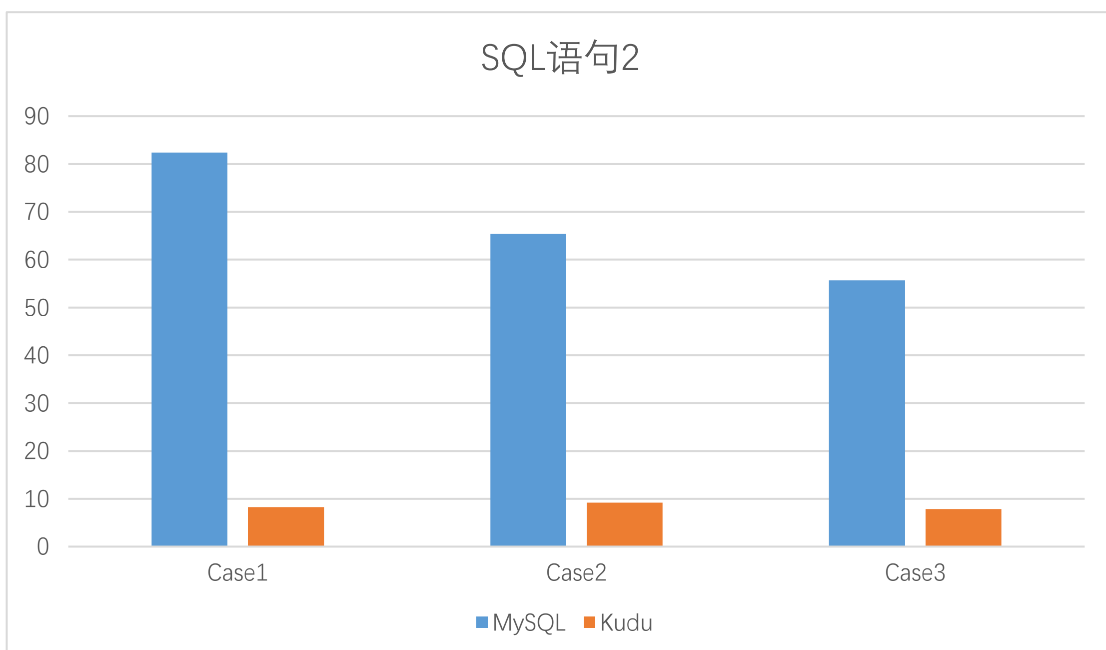


(2) 从领域相关表中提取出 1000 个与某领域最为相关的领域之间的相关关系

```
select FOSID as Source, FOSReferencesCount.FOSReference as
Target, Similarity/10000000 as Weight from
(select FOSReference
    from `FOSReferencesCount`
    where `FOSID` = '0271BC14'
    order by `Similarity` desc
    limit 1000) e1,
(select FOSReference
    from `FOSReferencesCount`
    where `FOSID` = '0271BC14'
    order by `Similarity` desc
    limit 1000) e2,
FOSReferencesCount
where e1.`FOSReference` = `FOSReferencesCount`.FOSID and
e2.`FOSReference` = `FOSReferencesCount`.FOSReference;
```

使用到的 Table 为 FOSReferencesCount，包含的记录数为 108374649 条。
测试结果如下：

FOSID	Computer Science (0271BC14)	Ethnic studies (03D2C4FF)	Data Structure (09ACCB7D)
MySQL	82.4s	65.4s	55.7s
Kudu	8.23s	9.175s	7.821s



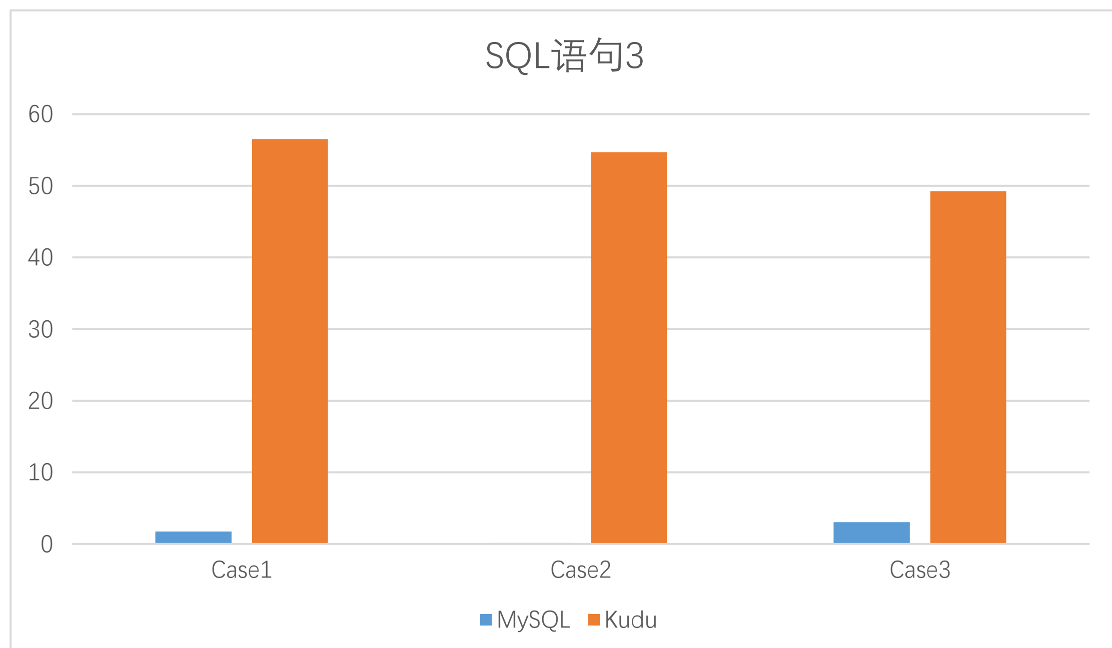
(3) 统计一个作者的 SCI 引用数

```
SELECT    count(*),SUM(SCICitation) as sum from
                                                      PaperSciReferencesCount CROSS JOIN
                                                      (select PaperID
                                                              from PaperAuthorAffiliations
                                                              where AuthorID = '0000194E' )
  AS TB1 on PaperSciReferencesCount.PaperReferenceID = TB1.PaperID
```

使用到的 Table 为 PaperSciReferencesCount 和 PaperAuthorAffiliations，PaperSciReferencesCount 中的记录数为 25370939 条，PaperAuthorAffiliations 中的记录数为 338222414 条。

测试结果如下

AuthorID	Fredrik Vult von Steyern (0000194E)	S Haykim (7A213E4C)	Robert H Thomas (7D63F748)
MySQL	1.766s	0.140s	3.062s
Kudu	56.544s	54.677s	49.260s



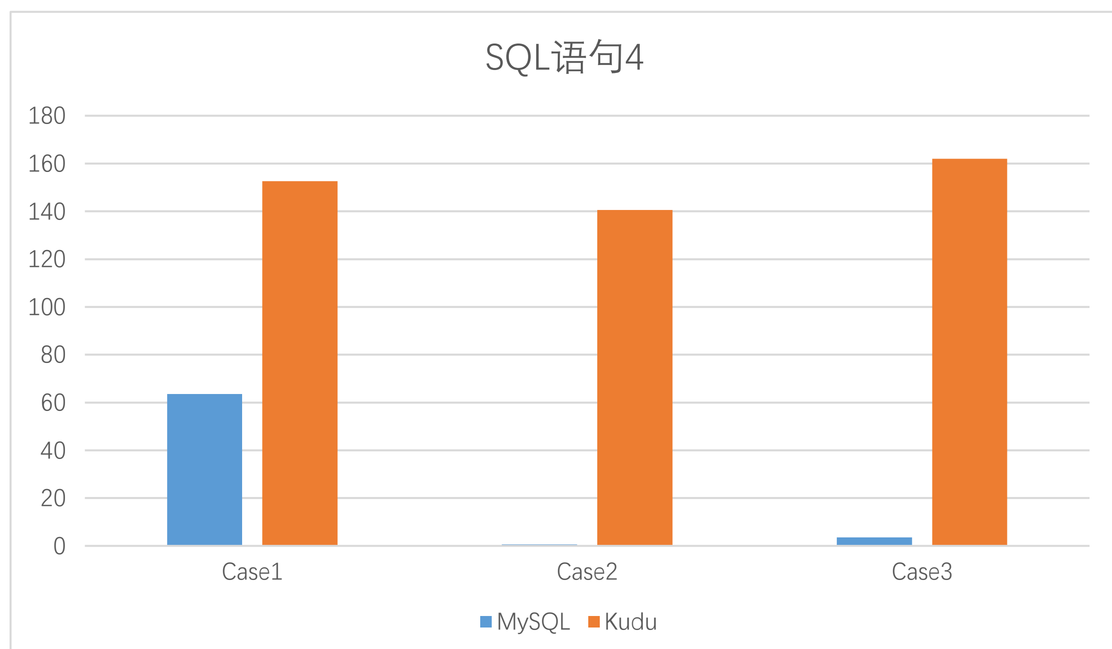
(4) 找出每篇论文被某个领域引用的数目

```
SELECT FieldsOfStudyID,FieldsOfStudyName,FieldCitation
from FieldsOfStudy
INNER JOIN (
    select FieldOfStudyIDMappedToKeyword,COUNT(*) as FieldCitation
    from PaperKeywords
    INNER JOIN
    (select PaperID
        from PaperReferences
        where PaperReferenceID = '786E45C5') as TB1
    on TB1.PaperID = PaperKeywords.PaperID
    GROUP by FieldOfStudyIDMappedToKeyword) as TB2
on TB2.FieldOfStudyIDMappedToKeyword = FieldsOfStudy.FieldsOfStudyID
order by FieldCitation desc limit 25;
```

使用到的 Table 有 FieldsOfStudy、PaperKeywords 和 PaperReferences，FieldsOfStudy 中的记录数为 53834 条，PaperKeywords 中的记录数为 163893049，PaperReferences 中的记录数为 536595630 条。

测试结果：

PaperReferenceID	786E45C5	03E93FD0	810C0047
MySQL	63.5s	0.656s	3.594s
Kudu	152.669s	140.594s	162.064s



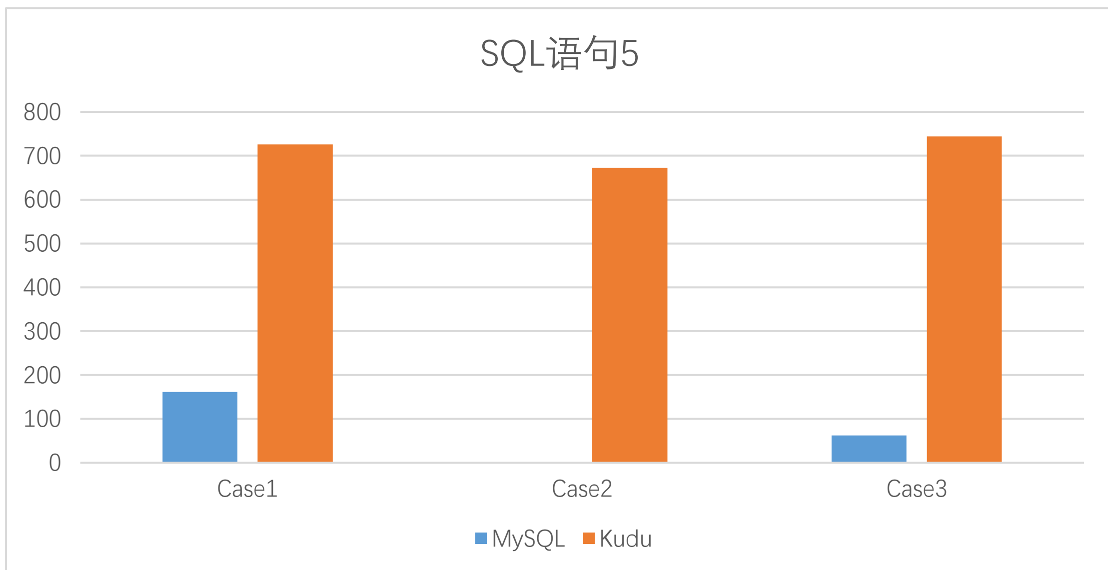
(5) 寻找引用某作者的论文及改论文引用该作者的次数

```
select PaperReferences.PaperID,count(*) from PaperReferences inner join
    (select * from PaperAuthorAffiliations where AuthorID='0C7733DB') as TB
    on PaperReferences.PaperReferenceID = TB.PaperID
    group by PaperReferences.PaperID
```

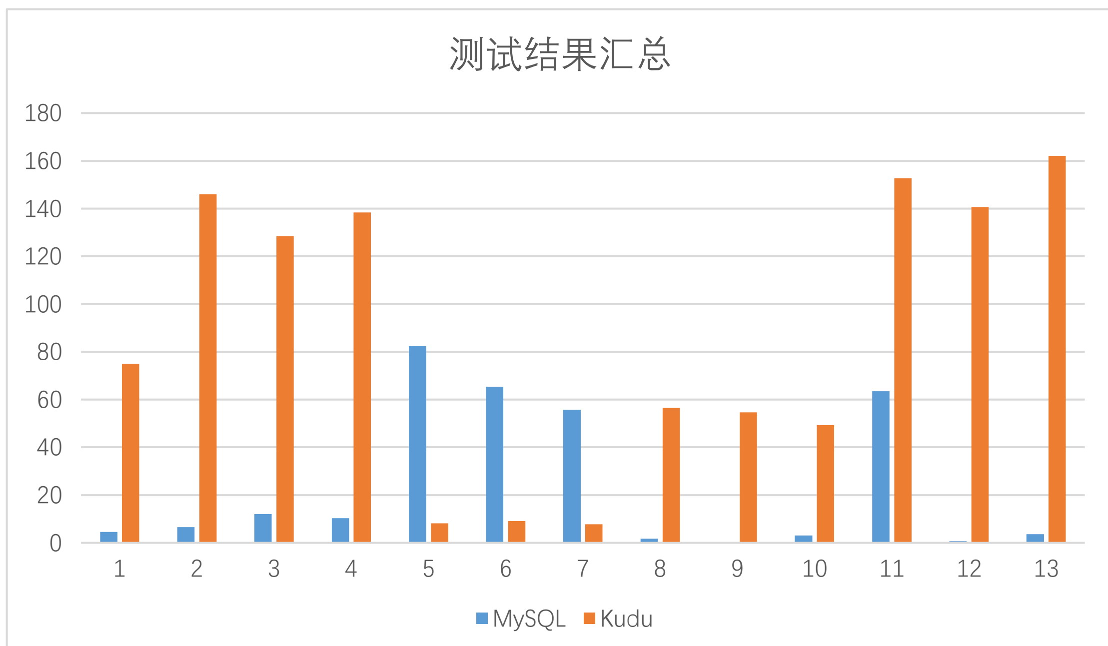
使用到的 Table 有 PaperReferences 和 PaperAuthorAffiliations。PaperReferences 中的记录数为 536595630 条，PaperAuthorAffiliations 中的记录数为 338222414 条。

测试结果：

AuthorID	j j hopfield 0C7733DB	Robert H Thomas 7D63F748	richard p lippmann 7F62A991
MySQL	161.3s	1.344s	62.375s
Kudu	725.959s	672.594s	744.367s



上述几组测试结果汇总后得到下图(为方便观测暂时排除了最后一组数据):



七、 测试结果分析

进行了几个 SQL 语句的测试后对结果有点失望，按理说使用了分布式数据库后查询速度应该有所提升，但在几组测试结果中 Kudu 的速度比 MySQL 慢了不少，耗时大概是 MySQL 的 20~30 倍。不过还是有一组结果 Kudu 的表现比 MySQL 优秀不少。我分析可能的原因有以下几个：

（1） Kudu 的集群中使用到了 Slave3 这台机器，而 Slave3 的内存和性能比另外两台机器差了不少，可能拖累了整个集群的性能。

（2） SQL 语句查询用的是 Spark 的接口，而一般来说 Kudu 是配合 Impala 使用的。有可能 Impala 与 Kudu 的适应性更好，用 Impala 进行查询速度会更快。

（3） Kudu 的表结构设计应该有许多讲究，比如说表的分区。在创建表的时候我尝试过使用 Hash Partition 和 Range Partition 两种不同的分区方式。两种分区方式在 SQL 测试中的结果有不小的差距，使用 Range Partition 的 Table 查询速度更快。所以我认为合理地设计表结构才能充分发挥出 Kudu 的性能，而我之前并没有针对这方面进行优化。

（4） 两种工具的应用场景有所不同。这几组查询的结果差异还是挺大的，既出现了 Kudu 比 MySQL 慢很多的情况，也出现了 Kudu 比 MySQL 快不少的情况。本身 Kudu 的定位是“Fast Analytics on Fast Data”，所以对于不同的场景来说 Kudu 的性能可能会有所差异，可能测试的这些 SQL 查询并不是 Kudu 所擅长的。此外，这次只测试了查询的速度，但其实对数据库来说，增删查改都是基本的操作，都算数据库性能评判的指标，所以以上的结果还不足以全面地说明 Kudu 的性能。

八、 总结

Kudu 作为一个新兴的分布式存储系统，相关的资料并不多，应用也并不广。就我所知在国内，小米公司是使用了 Kudu+Impala 来解决实时数据的 ad-hoc 查询需求，帮 Cloudera 在生产环境验证 Kudu。所以其实 Kudu 目前还并不成熟，暂时还不适合部署到实际的生产环境。在我这次尝试的过程中 Kudu 也并没有展现出特别优秀的性能，所以可能 Kudu 的应用场景并不适合我们 Acemap 的这种场景，不适合作为一种解决方案部署到我们的 Acemap。不过在我使用的过程中，觉得 Kudu 一个特别大的优点就是与 Hadoop 生态中的各种工具友好集成，提供了丰富的接口，未来等 Kudu 更加成熟完善之后应该会得到更广的应用。