**电商系统数据库存取分析**

夏文轩 3110000104 许是程 3100000210 方牧心 3110000155

1. **系统简介**

随着电子商务的发展，电商系统的数据库规模也越来越大，逐步步入了一个大数据时代，如何能够合理的存储和分析数据成了几大电商的关注重点。以淘宝网为例，在2012年淘宝“双11”购物节的时候，支付宝交易总额达到191亿元，交易单量达到1亿笔。如此巨大的成交量背后是海量数据的存取与交换，如何能够提高存取与交换的效率，成了各大电商的研究重点。

此次我们小组便以电商系统的数据库存取为例，分析在几种体系框架下存取性能的表现，来研究体系框架对系统性能的影响。

1. **Node.Js框架分析**

Node.Js是基于V8虚拟机，可构建高性能服务器的语言。通过单线程的轮寻，达到并发的效果，而不像php一样，对于每一个访问都创造一个线程。所以，Node.Js是一个提供高性能而降低复杂程度的服务器框架部分。

相对于电商系统中的质量属性，我们有如下的分析：

1. 大量的并发： Node.Js相对于php，更支持大量的并发。因为php是对于每一个连接创立线程，保存在服务器端。而每一个线程，都需要一定的内存。一旦服务器的访问量过大，那内存的占用率会非常的高。而对于Node.Js来说，它是单线程的，根本不存在这样的问题。

2. 储存空间： 这并不是Node.Js能改善的。这里并不作讨论。

3. 高速的用户体验：Node.Js虽然没有在访问的时候创建线程。而server端的阻塞是服务器遇到的最大问题。一方面，访问时加入的新进程，使得老进程的处理变得异常缓慢。这样，一些长时间进程容易饿死，造成用户体验的下降。而Node.Js并不存在这样的问题：因为Node.Js是通过轮寻来处理并发的（我提到了这点多次，这是我选择Node.js的最重要原因）。

4. 安全性在这里并不讨论。

5. 鲁棒性：在我们配置使用windows作为服务器系统。而windows用的是cmd命令行来驱使nodejs服务运行。一旦服务由于故障断裂，cmd就会自动断开报错。所以，这样的状况下，鲁棒性并不好。而改进措施是，在服务器本地写一个脚本，来监视服务的运行，一旦断开，则重启，并保存日志。

1. **Memcached框架分析**

Memcached是一个高性能、分布式内存对象缓存系统，主要用于通过减少数据库调用来加速动态网页应用。由LiveJournal的Brad Fitzpatrick开发，但目前被许多网站使用，如Digg, Facebook, Wikipedia等。

Memcached允许你把系统中的冗余内存分配给内存紧缺的部分使用，使你能更好地使用内存。考虑一个分布式服务器，如果没有Memcached，则每个节点只能使用其自身的内存，而Memcached则允许一个节点使用其他节点的内存。

其具有以下特点：

1. 协议简单：memcached的服务器客户端通信并不使用复杂的MXL等格式，而是使用简单的基于文本的协议。
2. 基于libevent的事件处理：libevent是个程序库，他将Linux 的epoll、BSD类操作系统的kqueue等时间处理功能封装成统一的接口。memcached使用这个libevent库，因此能在Linux、BSD、Solaris等操作系统上发挥其高性能。
3. 内置内存存储方式：为了提高性能，memcached中保存的数据都存储在memcached内置的内存存储空间中。由于数据仅存在于内存中，因此重启memcached，重启操作系统会导致全部数据消失。另外，内容容量达到指定的值之后memcached回自动删除不适用的缓存。
4. Memcached不互通信的分布式：memcached尽管是“分布式”缓存服务器，但服务器端并没有分布式功能。各个memcached不会互相通信以共享信息。他的分布式主要是通过客户端实现的。

Memcached删除数据时数据不会真正从memcached中消失。Memcached不会释放已分配的内存。记录超时后，客户端就无法再看见该记录（invisible 透明），其存储空间即可重复使用。

Lazy Expriationmemcached内部不会监视记录是否过期，而是在get时查看记录的时间戳，检查记录是否过期。这种技术称为lazy expiration.因此memcached不会再过期监视上耗费CPU时间。

对于缓存存储容量满的情况下的删除需要考虑多种机制，一方面是按队列机制，一方面应该对应缓存对象本身的优先级，根据缓存对象的优先级进行对象的删除。

1. **Hadoop框架分析**

Hadoop是一个开发和运行处理大规模数据的软件平台，是Appach的一个用java语言实现开源软件框架，实现在大量计算机组成的集群中对海量数据进行分布式计算。

Hadoop框架中最核心设计就是：HDFS和MapReduce。HDFS提供了海量数据的存储，MapReduce提供了对数据的计算。

HDFS: Hadoop Distributed File System，Hadoop的分布式文件系统。

大文件被分成默认64M一块的数据块分布存储在集群机器中。

MapReduce：Hadoop为每一个input split创建一个task调用Map计算，在此task中依次处理此split中的一个个记录(record)，map会将结果以key--value的形式输出，hadoop负责按key值将map的输出整理后作为Reduce的输入，Reduce Task的输出为整个job的输出，保存在HDFS上。

Hadoop的集群主要由 NameNode， DataNode， Secondary NameNode， JobTracker， TaskTracker组成，在此基础上运用分布式存储与分布式计算，可以显著提高存取和分析的效率和性能。

其优点有：

1. 可以处理超大文件

这里的超大文件通常是指百MB、甚至数百TB大小的文件。目前在实际应用中，HDFS已经能用来存储管理PB级的数据了。

1. 拥有流式的访问数据

HDFS的设计建立在“一次写入、多次读写”任务的基础上。这意味着一个数据集一旦由数据源生成，就会被复制分发到不同的存储节点中，然后响应各种各样的数据分析任务请求。在多数情况下，分析任务都会涉及数据集中的大部分数据，也就是说，对HDFS来说，请求读取整个数据集要比读取一条记录更加高效。

1. 运行于廉价的商用机器集群上

Hadoop设计对应急需求比较低，只须运行在低廉的商用硬件集群上，而无需在昂贵的高可用性机器上。廉价的商用机也就意味着大型集群中出现节点故障情况的概率非常高。HDFS遇到了上述故障时，被设计成能够继续运行且不让用户察觉到明显的中断。

可见Hadoop的特点使它可以成为一种理想的电商系统数据库的解决方案。

1. **示例程序**

我们以Hadoop系统为例做了一个Demo程序，来分析它与传统数据库的性能差异。

1. **数据获取**

我们需要大量的数据来验证我们的想法，于是开始寻找数据源。我们首先考虑的是Amazon，因为Amazon的数据比较权威，许多大的项目都使用Amazon的数据作为测试对象，可见其在学术领域，作为数据源的地位。但是遗憾的是，Amazon在防爬虫方面做了很大的功夫。对于一直刷页面的ID，给予了一定的保护，以至于我们第一个程序爬下来的只有一段<script>。而后，我们考虑了淘宝，但是淘宝的链接做的相当的杂乱，并不方便爬取我们需要的数据。所以再次放弃了淘宝作为数据源的想法。

我们最终选择的是京东作为数据源。原因有如下三点：1.京东作为国内知名的电商系统，拥有较为庞大的数据存储；2.京东在爬虫上不设防备（我也不知道原因）；3.京东的搜索页面较为规范，便于爬虫的进行。



这是一个京东的搜索结果页面，我们可以看到，搜索的条目非常清楚的呈现。

爬取过程：

我们使用了python写爬虫代码，使用了一个非常好用的开源库：BeautifulSoup作为网页HTML分析的工具插件。代码如下：

|  |
| --- |
| import os,urllib2  import string  from BeautifulSoup import BeautifulSoup  import csv, sys  url\_head = "http://search.jd.com/search?keyword=%D5%DC%D1%A7&rt=1&book=y&ev=&area=15&page="  url\_tail = ""  csvfile = file('D:/gruber/out109.csv', 'wb')  writer = csv.writer(csvfile)  for i in range(1, 599):  page\_str = str(i)  url = url\_head + page\_str + url\_tail  print(url)  flag = 1;  while flag:  try:  response = urllib2.urlopen(url)  html = response.read()  flag = 0;  except Exception, e:  pass;  # html = html.decode('gbk')  # html = html.encode('2312')  # print(html)  my\_soup = BeautifulSoup(html);  # print my\_soup.prettify();  items = my\_soup.findAll('div', {"class": "item"});  for item in items:  book\_name = ''  book\_price = ''  publisher = ''  book\_author = ''  publish\_time = ''  a\_link = ''  error = 0;  #name  my\_item = BeautifulSoup(str(item))  p\_name = my\_item.findAll("dt", {"class": "p-name"});  for name in p\_name:  book\_name = name.contents[1].contents[0]  try:  book\_name = book\_name.encode('gbk').strip();  book\_name = str(book\_name)  except Exception, e:  error = 1;    # book\_name = book\_name.encode('gbk')  # print(book\_name)  a\_link = name.contents[1]['href'];  a\_link = a\_link.encode('gbk');  # a\_link = str(a\_link)  # print(a\_link)  if (error == 1):  continue;  #price  p\_market = my\_item.findAll('dd', {"class": "p-market"});  for market in p\_market:  book\_price = market.contents[3].contents[0].contents[0];  book\_price = book\_price.encode('gbk')      # book\_price = str(book\_price)  # print(book\_price)  #SUMMARY  summary = my\_item.findAll("ul", {"class" : "summary"});  for smr in summary:  my\_smr = BeautifulSoup(str(smr));  summary\_press = my\_smr.findAll("li", {"class" : "summary-press"});  if (len(summary\_press) != 0):  publisher = summary\_press[0].contents[3].contents[0];  try:  publisher = publisher.contents[0];  except Exception, e:  pass;  publisher = publisher.encode('gbk')  # publisher = str(publish\_time)  # print(publisher)  summary\_time = my\_smr.findAll("li", {"class" : "summary-time"});  if (len(summary\_time) != 0):  publish\_time = summary\_time[0].contents[3].contents[0];  publish\_time = publish\_time.encode('gbk')  # publish\_time = str(publish\_time)  # print(publish\_time)  summary\_author = my\_smr.findAll("li", {"class" : "summary-author"})  # print(len(summary\_author))  if (len(summary\_author) != 0):  author = summary\_author[0].contents[3]  author\_a = BeautifulSoup(str(author))  author\_q = author\_a.findAll('a');  book\_author = author\_q[0].contents[0];  book\_author = book\_author.encode('gbk')  writer.writerow([book\_name, book\_price, publisher, book\_author, publish\_time, a\_link]) |

这里，每一次爬取，都需要手工输入一个url\_head，作为爬取页面的首页。爬虫程序会根据URL，自动翻页，爬取一个搜索词汇的所有书籍的条目，并以.csv的形式存储在本地。

我们保存的item有：书名，价格，出版社，出版日期，链接。



缺点：没能实现一个对于一个list关键词的爬虫，而是要每一次输入一个搜索关键词，效率比较低下。

我们一共爬取了100,000条书籍数据，作为我们的测试数据。

1. **Hadoop环境配置**
2. 实验平台

主机：Fedora 19 x86\_64

虚拟机：Ubuntu 13.10 x86\_64

JAVA 1.7.0\_55

Hadoop 1.2.1

主机IP：192.168.56.1 NameNode, JobTracker

slave1：192.168.56.101 DataNode, TaskTracker

slave2：192.168.56.102 DataNode, TaskTracker

slave3：192.168.56.103 DataNode, TaskTracker

1. 主机上的准备

修改Hosts

vi /etc/hosts

192.168.56.1 master

192.168.56.101 slave1

192.168.56.102 slave2

192.168.56.103 slave3

修改hostname为master：

# /etc/hostname

master

1. 安装JAVA

从Oracle官网下载JAVA SDK：

http://www.oracle.com/technetwork/java/javase/downloads/index.html

安装JDK包：sudo rpm -Uvh /path/to/binary/jdk-7u55-linux-x64.rpm

选择使用Oracle JDK：

## java ##

sudo alternatives --install /usr/bin/java java /usr/java/latest/jre/bin/java 200000

## javaws ##

sudo alternatives --install /usr/bin/javaws javaws /usr/java/latest/jre/bin/javaws 200000

## Java Browser (Mozilla) Plugin 64-bit ##

sudo alternatives --install /usr/lib64/mozilla/plugins/libjavaplugin.so libjavaplugin.so.x86\_64 /usr/java/latest/jre/lib/amd64/libnpjp2.so 200000

## Install javac only if you installed JDK (Java Development Kit) package ##

sudo alternatives --install /usr/bin/javac javac /usr/java/latest/bin/javac 200000

alternatives --install /usr/bin/jar jar /usr/java/latest/bin/jar 200000

1. 安装Hadoop

下载Hadoop：http://mirrors.hust.edu.cn/apache/hadoop/common/

建立hadoop用户：useradd hadoop

建立hadoop文件夹的symbolink：ln -s ~/bin/hadoop-1.2.1 ~/bin/hadoop

把Hadoop解压到~/bin目录下，并更新~/.bashrc

# ~/.bashrc

# hadooop

export JAVA\_HOME="/usr/java/latest"

export HADOOP\_INSTALL="/home/hadoop/bin/hadoop"

export PATH=$PATH:$HADOOP\_INSTALL/bin:$HADOOP\_INSTALL/sbin

export CLASSPATH="/home/hadoop/bin/mysql-connector-java-bin.jar"

1. 配置Hadoop

在hadoop-env.sh中需指定JAVA安装路径（JAVA\_HOME)

# ~/bin/hadoop/conf/hadoop-env.sh

export JAVA\_HOME=/usr/java/latest

在core-site.xml中需指定HDFS的端口号(fs.default.name)和根目录(hadoop.tmp.dir)，这里端口号设为8020，根目录设为~/hadoop\_root

# ~/bin/hadoop/conf/core-site.xml

<?xml version="1.0"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>

<name>fs.default.name</name>

<value>hdfs://master:8020</value>

</property>

<property>

<name>hadoop.tmp.dir</name>

<value>/home/hadoop/hadoop\_root</value>

</property>

</configuration>

在mapred-site.xml中需要设定Mapreduce Job Tracker端口(mapred.job.tracker)和Map reduce的临时文件目录()

# ~/bin/hadoop/conf/mapred-site.xml

<?xml version="1.0"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>

<name>mapred.job.tracker</name>

<value>http://master:8021</value>

</property>

<property>

<name>mapred.local.dir</name>

<value>/home/hadoop/hadoop\_root/mapred</value>

</property>

</configuration>

在hdfs-site.xml中需设定NameNode持久存储名字空间及事务日志的本地文件系统路径(dfs.name.dir)，HDFS数据路径(dfs.data.dir)和数据需要备份的数量(dfs.replication)，其中备份数量不能超过节点数。

# ~/bin/hadoop/conf/hdfs-site.xml

<?xml version="1.0"?>

<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<property>

<name>dfs.name.dir</name>

<value>/home/hadoop/hadoop\_root/name</value>

<description> </description>

</property>

<property>

<name>dfs.data.dir</name>

<value>/home/hadoop/hadoop\_root/hdfs</value>

<description> </description>

</property>

<property>

<name>dfs.replication</name>

<value>3</value>

</property>

</configuration>

最后配置master和slaves的机器名，即在hosts中写的名字。

~/bin/hadoop/conf/masters

master

~/bin/hadoop/conf/slaves

master

slave1

slave2

slave3

1. SSH配置

开启SSH服务：sudo service sshd start

使主机可以不用密码SSH登录虚拟机：cat ~/.ssh/id\_rsa.pub

将终端输出的内容拷贝到虚拟机中的~/.ssh/authorized\_keys

尝试连接虚拟机：ssh 192.168.56.101

1. 虚拟机上的准备

修改Hosts

vi /etc/hosts

192.168.56.1 master

192.168.56.101 slave1

192.168.56.102 slave2

192.168.56.103 slave3

1. 安装JAVA

sudo add-apt-repository ppa:webupd8team/java

sudo apt-get update

sudo apt-get install oracle-java7-installer

1. 安装SSH

sudo apt-get install ssh

1. 安装Hadoop

直接把本机上的Hadoop拷贝过去： scp -r bin/ slave1:~/ # 在本机上执行

更新~/.bashrc，和主机唯一的差别是JAVA\_HOME。

# hadooop

export JAVA\_HOME="/usr/lib/jvm/java-7-oracle/"

export HADOOP\_INSTALL="/home/hadoop/bin/hadoop"

export PATH=$PATH:$HADOOP\_INSTALL/bin:$HADOOP\_INSTALL/sbin

同理，hadoop\_env.sh中也需要相应修改JAVA\_HOME。

准备完成后备份虚拟机文件，以防止后续配置出错而导致需要重装。然后将虚拟机文件拷贝两份。然后需要在/etc/hostname中修改host名称为对应的slave1, slave2或slave3。

1. 启动Hadoop

格式化一个新的文件系统：

[hadoop@localhost ~]$ hadoop namenode -format

14/04/26 17:35:02 INFO namenode.NameNode: STARTUP\_MSG:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

STARTUP\_MSG: Starting NameNode

STARTUP\_MSG: host = localhost.localdomain/127.0.0.1

STARTUP\_MSG: args = [-format]

STARTUP\_MSG: version = 1.2.1

STARTUP\_MSG: build = https://svn.apache.org/repos/asf/hadoop/common/branches/branch-1.2 -r 1503152; compiled by 'mattf' on Mon Jul 22 15:23:09 PDT 2013

STARTUP\_MSG: java = 1.7.0\_51

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

14/04/26 17:35:02 INFO util.GSet: Computing capacity for map BlocksMap

14/04/26 17:35:02 INFO util.GSet: VM type = 64-bit

14/04/26 17:35:02 INFO util.GSet: 2.0% max memory = 932184064

14/04/26 17:35:02 INFO util.GSet: capacity = 2^21 = 2097152 entries

14/04/26 17:35:02 INFO util.GSet: recommended=2097152, actual=2097152

14/04/26 17:35:03 INFO namenode.FSNamesystem: fsOwner=hadoop

14/04/26 17:35:03 INFO namenode.FSNamesystem: supergroup=supergroup

14/04/26 17:35:03 INFO namenode.FSNamesystem: isPermissionEnabled=true

14/04/26 17:35:03 INFO namenode.FSNamesystem: dfs.block.invalidate.limit=100

14/04/26 17:35:03 INFO namenode.FSNamesystem: isAccessTokenEnabled=false accessKeyUpdateInterval=0 min(s), accessTokenLifetime=0 min(s)

14/04/26 17:35:03 INFO namenode.FSEditLog: dfs.namenode.edits.toleration.length = 0

14/04/26 17:35:03 INFO namenode.NameNode: Caching file names occuring more than 10 times

14/04/26 17:35:03 INFO common.Storage: Image file /home/hadoop/hadoop\_root/name/current/fsimage of size 112 bytes saved in 0 seconds.

14/04/26 17:35:03 INFO namenode.FSEditLog: closing edit log: position=4, editlog=/home/hadoop/hadoop\_root/name/current/edits

14/04/26 17:35:03 INFO namenode.FSEditLog: close success: truncate to 4, editlog=/home/hadoop/hadoop\_root/name/current/edits

14/04/26 17:35:04 INFO common.Storage: Storage directory /home/hadoop/hadoop\_root/name has been successfully formatted.

14/04/26 17:35:04 INFO namenode.NameNode: SHUTDOWN\_MSG:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

SHUTDOWN\_MSG: Shutting down NameNode at localhost.localdomain/127.0.0.1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

启动Hadoop集群：start-all.sh

注意这里不能使用ctrl + C强制退出，否则会集群会陷入Safe Mode。

其他错误解析：

INFO org.apache.hadoop.mapred.TaskTracker: Failed to get system directory...

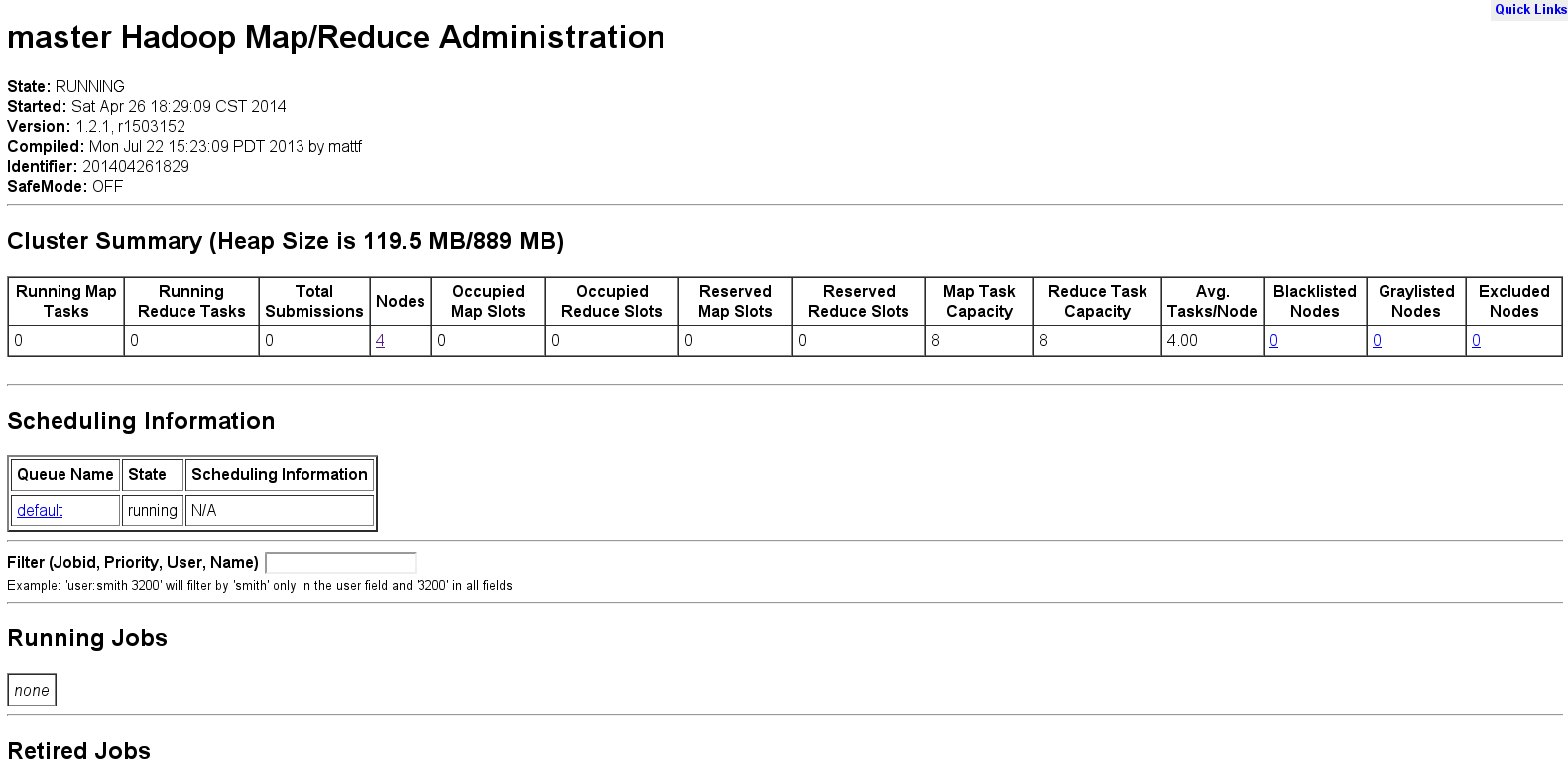
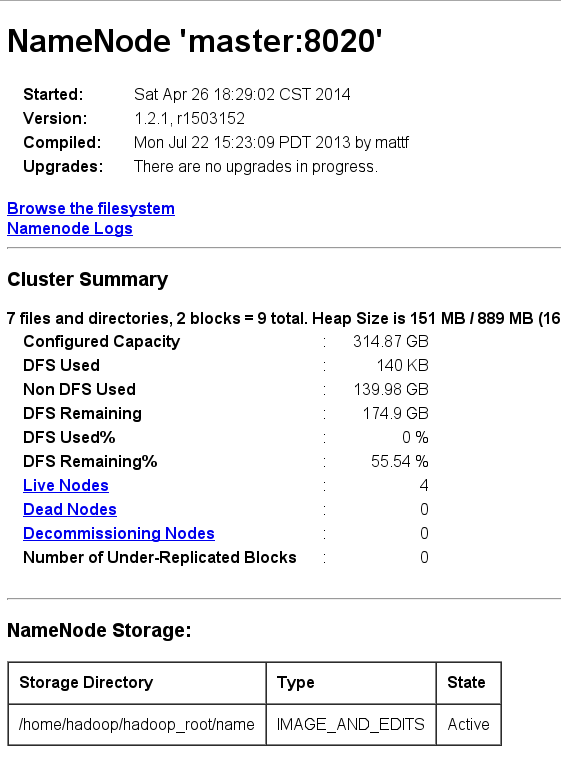
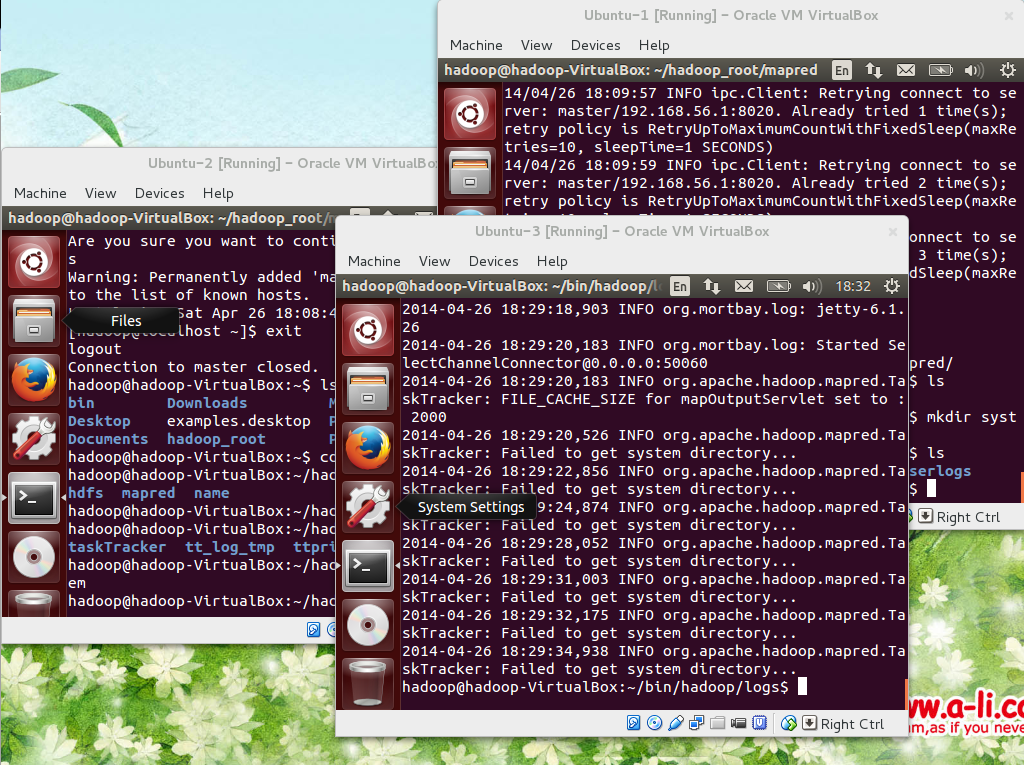
需要在mapred目录下创建system目录。（需要在每个node执行）

Can not start task tracker because java.io.IOException: Call to master/192.168.56.1:8021 failed on local exception: java.net.NoRouteToHostException: No route to host

端口被防火墙关闭了，可以选择增加防火墙规则或者关闭防火墙。

成功后可以访问Name node master页面http://master:50070和JobTracker页面http://master:50030。

截图如下所示



1. 运行Hadoop样例程序计算Pi

[hadoop@master hadoop]$ hadoop jar hadoop-examples-1.2.1.jar pi 3 10

Number of Maps = 3

Samples per Map = 10

Wrote input for Map #0

Wrote input for Map #1

Wrote input for Map #2

Starting Job

14/04/26 22:06:57 INFO mapred.FileInputFormat: Total input paths to process : 3

14/04/26 22:06:57 INFO mapred.JobClient: Running job: job\_201404262204\_0003

14/04/26 22:06:58 INFO mapred.JobClient: map 0% reduce 0%

14/04/26 22:07:06 INFO mapred.JobClient: map 33% reduce 0%

14/04/26 22:07:08 INFO mapred.JobClient: map 66% reduce 0%

14/04/26 22:07:10 INFO mapred.JobClient: map 100% reduce 0%

14/04/26 22:07:14 INFO mapred.JobClient: map 100% reduce 33%

14/04/26 22:07:15 INFO mapred.JobClient: map 100% reduce 100%

14/04/26 22:07:16 INFO mapred.JobClient: Job complete: job\_201404262204\_0003

14/04/26 22:07:16 INFO mapred.JobClient: Counters: 31

14/04/26 22:07:16 INFO mapred.JobClient: Job Counters

14/04/26 22:07:16 INFO mapred.JobClient: Launched reduce tasks=1

14/04/26 22:07:16 INFO mapred.JobClient: SLOTS\_MILLIS\_MAPS=17991

14/04/26 22:07:16 INFO mapred.JobClient: Total time spent by all reduces waiting after reserving slots (ms)=0

14/04/26 22:07:16 INFO mapred.JobClient: Total time spent by all maps waiting after reserving slots (ms)=0

14/04/26 22:07:16 INFO mapred.JobClient: Rack-local map tasks=2

14/04/26 22:07:16 INFO mapred.JobClient: Launched map tasks=3

14/04/26 22:07:16 INFO mapred.JobClient: Data-local map tasks=1

14/04/26 22:07:16 INFO mapred.JobClient: SLOTS\_MILLIS\_REDUCES=9593

14/04/26 22:07:16 INFO mapred.JobClient: File Input Format Counters

14/04/26 22:07:16 INFO mapred.JobClient: Bytes Read=354

14/04/26 22:07:16 INFO mapred.JobClient: File Output Format Counters

14/04/26 22:07:16 INFO mapred.JobClient: Bytes Written=97

14/04/26 22:07:16 INFO mapred.JobClient: FileSystemCounters

14/04/26 22:07:16 INFO mapred.JobClient: FILE\_BYTES\_READ=72

14/04/26 22:07:16 INFO mapred.JobClient: HDFS\_BYTES\_READ=714

14/04/26 22:07:16 INFO mapred.JobClient: FILE\_BYTES\_WRITTEN=223075

14/04/26 22:07:16 INFO mapred.JobClient: HDFS\_BYTES\_WRITTEN=215

14/04/26 22:07:16 INFO mapred.JobClient: Map-Reduce Framework

14/04/26 22:07:16 INFO mapred.JobClient: Map output materialized bytes=84

14/04/26 22:07:16 INFO mapred.JobClient: Map input records=3

14/04/26 22:07:16 INFO mapred.JobClient: Reduce shuffle bytes=84

14/04/26 22:07:16 INFO mapred.JobClient: Spilled Records=12

14/04/26 22:07:16 INFO mapred.JobClient: Map output bytes=54

14/04/26 22:07:16 INFO mapred.JobClient: Total committed heap usage (bytes)=623648768

14/04/26 22:07:16 INFO mapred.JobClient: CPU time spent (ms)=2930

14/04/26 22:07:16 INFO mapred.JobClient: Map input bytes=72

14/04/26 22:07:16 INFO mapred.JobClient: SPLIT\_RAW\_BYTES=360

14/04/26 22:07:16 INFO mapred.JobClient: Combine input records=0

14/04/26 22:07:16 INFO mapred.JobClient: Reduce input records=6

14/04/26 22:07:16 INFO mapred.JobClient: Reduce input groups=6

14/04/26 22:07:16 INFO mapred.JobClient: Combine output records=0

14/04/26 22:07:16 INFO mapred.JobClient: Physical memory (bytes) snapshot=638951424

14/04/26 22:07:16 INFO mapred.JobClient: Reduce output records=0

14/04/26 22:07:16 INFO mapred.JobClient: Virtual memory (bytes) snapshot=2835275776

14/04/26 22:07:16 INFO mapred.JobClient: Map output records=6

Job Finished in 19.799 seconds

Estimated value of Pi is 3.60000000000000000000



1. **Ruby on rails实验平台搭建**
2. 实验平台：

主机：Fedora 19 x86\_64

Ruby版本：2.0.0p353

Rails版本：4.0.2

Bootstrap版本：3.1.1

1. 网站功能

模型

网站主要包括三个模型，用户、书籍和购买记录。其中用户模型利用了devise插件，并添加了姓名信息；书籍模型的定义如下：

Books = {

title: string,

author: string,

publisher: string,

publish\_date: date,

price: float,

amazon\_link: string

}

注：amazon\_link即对应的书籍网页编号。

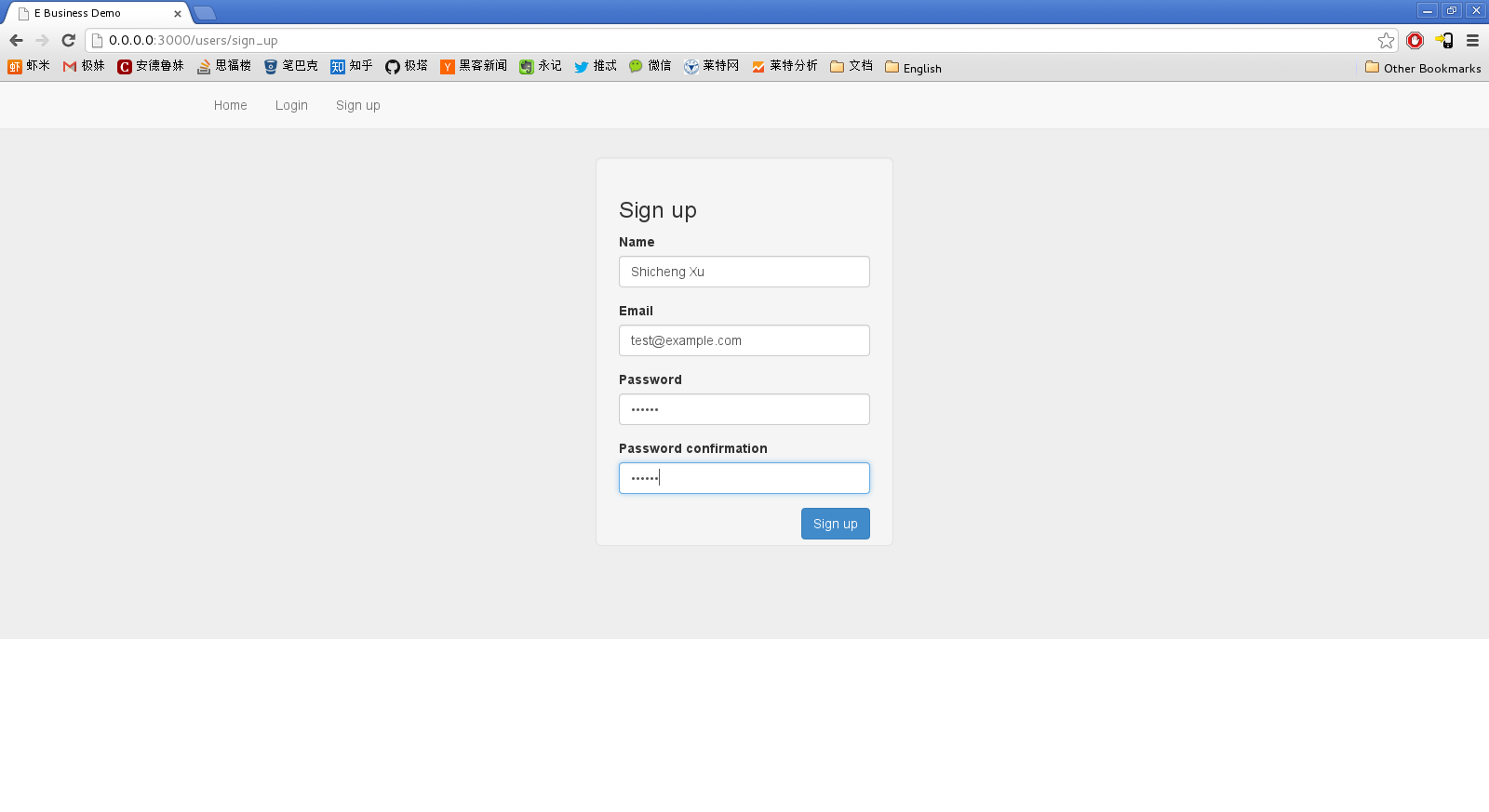
购买记录里面包含用户id、书籍id和购买时间。

功能

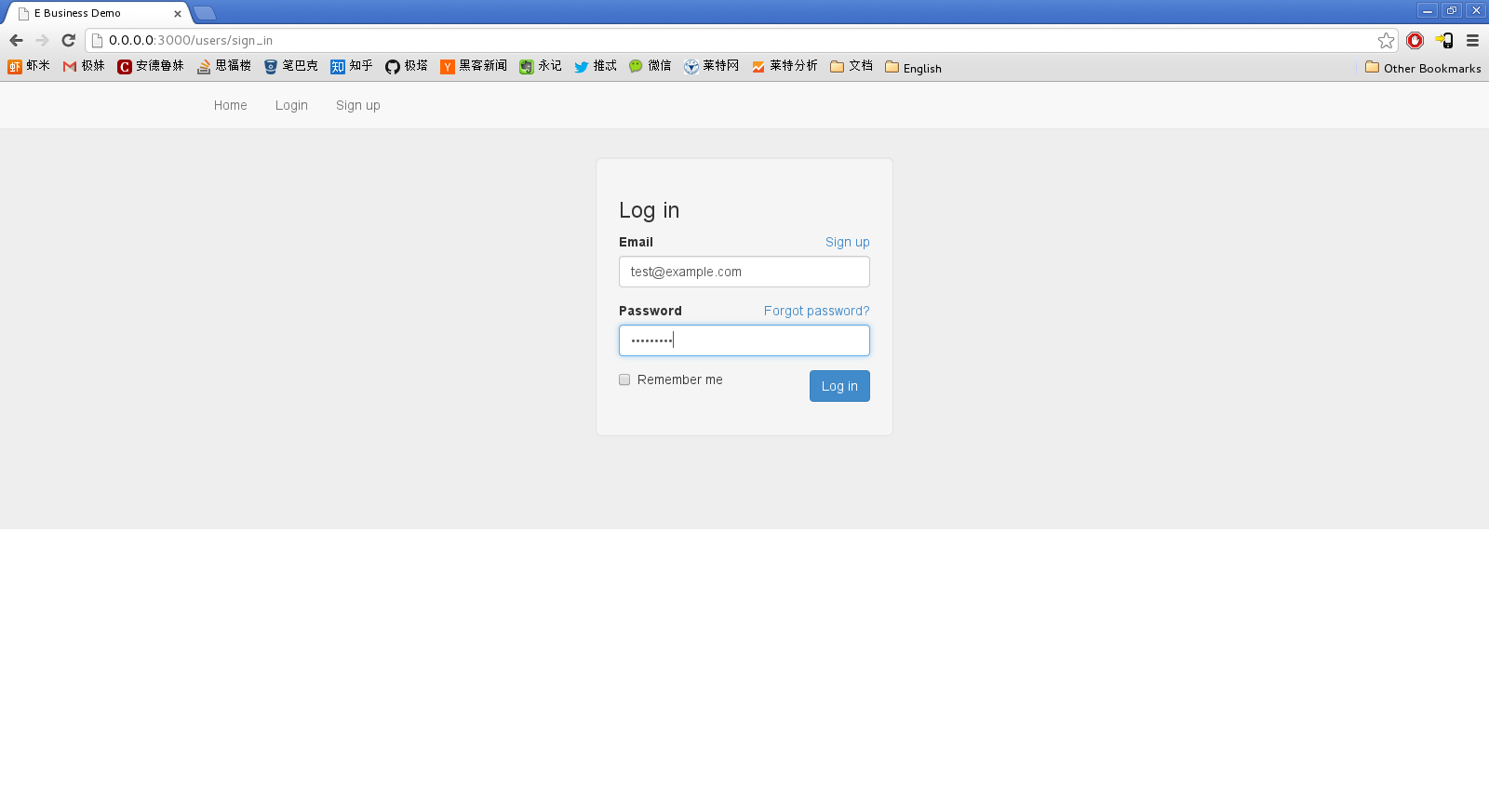
网站主要功能是用户注册登录，以及对书籍的创建、查看、更新和删除功能。

以下是网站的部分截图：

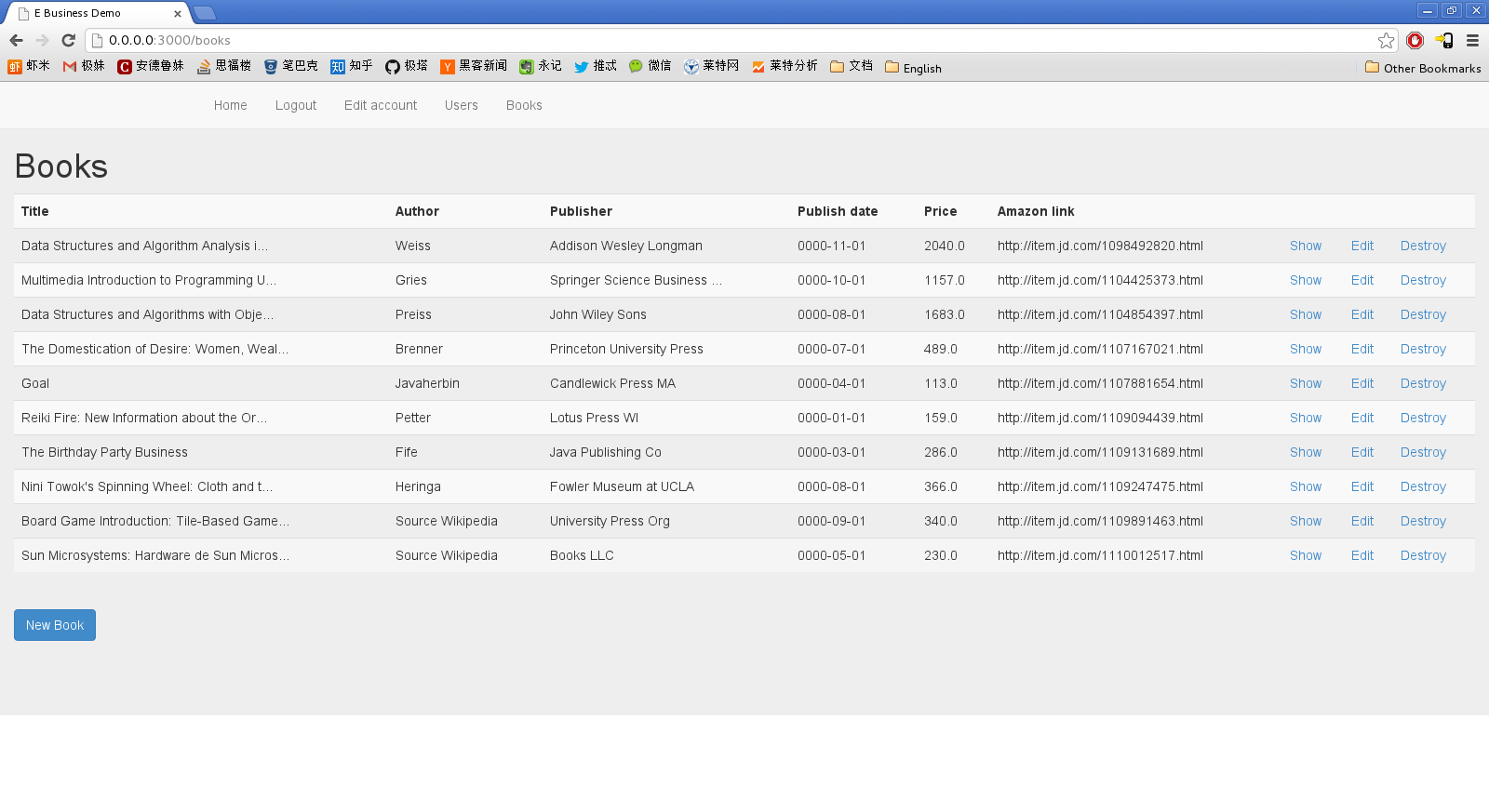
用户注册：



用户登录：



书籍展示页面：



1. **实验设计**

为了比较在电子商务系统服务器后端配置的传统数据库，与hadoop分布式存储结构的性能。我们进行了如下的实验：

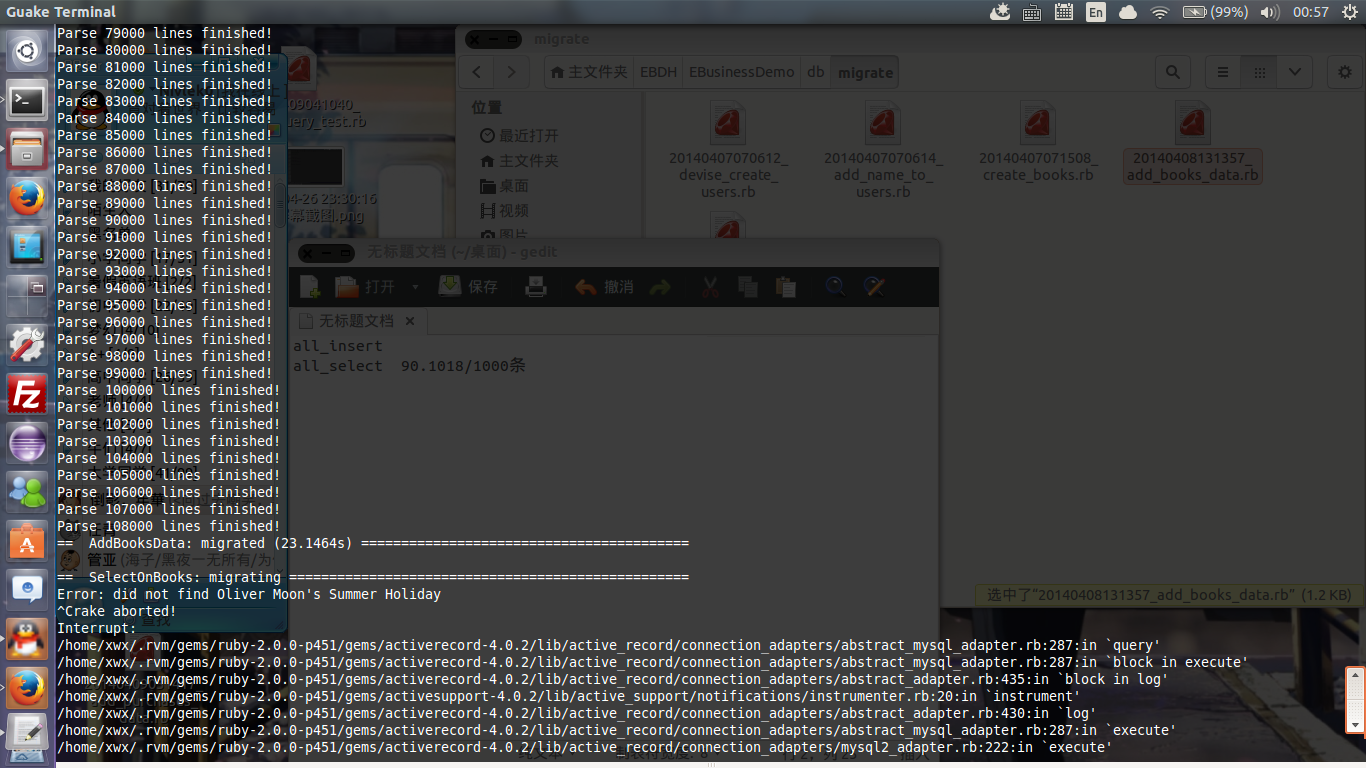
1. 将爬虫获取的100,000条书目数据逐一插入到mysql数据库表中，记录其花费的时间。
2. 我们粗略地认为，用户通常所查询的数据（热门数据），占所有数据的10%。在这里，我们的总数据量是100,000，所以我们粗略地认为，热门数据占10,000条。为了节约实验时间，我们在这10,000条热门数据中，查找其热门的1,000条数据，记录其花费的时间，算出平均查找时间。
3. 为了比较和传统数据库插入的效率，我们将原始的100,000条数据分为两类，一类为热门的10,000条数据，另一类为较为冷门的数据。我们将热门的数据放在服务器的数据库上，而将冷门的数据存放在集群上。我们记录这样的数据存储所花费的时间。
4. 为了比较热门数据查找的效率。我们在配置hadoop的系统上，查找热门数据。但为了和上述实验（2）做对照，我们同样选取了10,000条数据中的1,000条数据，作为查找对象，记录所花费的时间。
5. 对于将冷门上传到Hadoop集群的部分，我们采用了分块存储的部分，即每1000、3000或10000条记录存在一个文件中，以方便之后的存取。

Mysql插入查询实验结果：

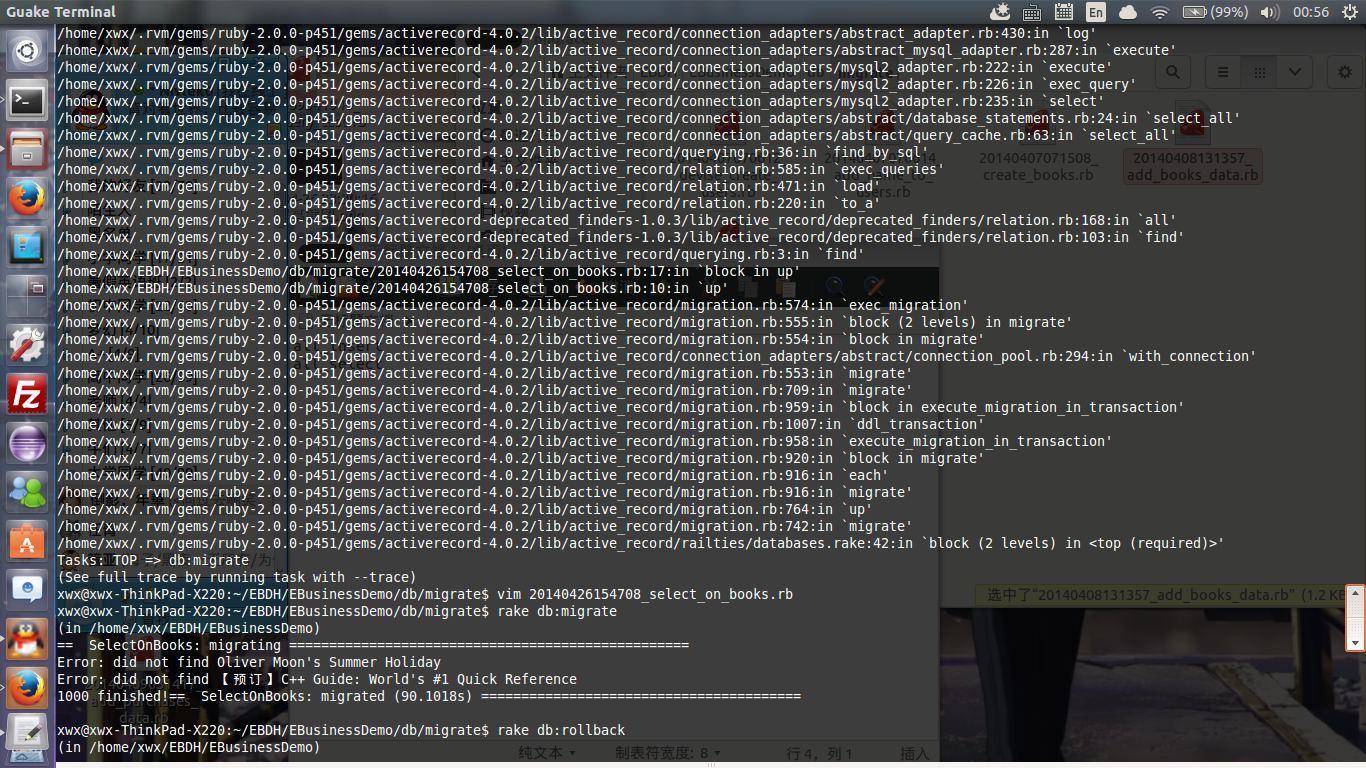
|  |  |  |
| --- | --- | --- |
|  | 传统Mysql | 带Hadoop分布存储mysql |
| 插入100,000条数据花费时间(s) | 23 | 2.1 |
| 查找1,000条数据花费时间(s) | 90 | 2.1 |
| 平均查找1条数据花费时间(s) | 0.09 | 0.002 |

上传数据至Hadoop集群实验结果（共99000条数据）

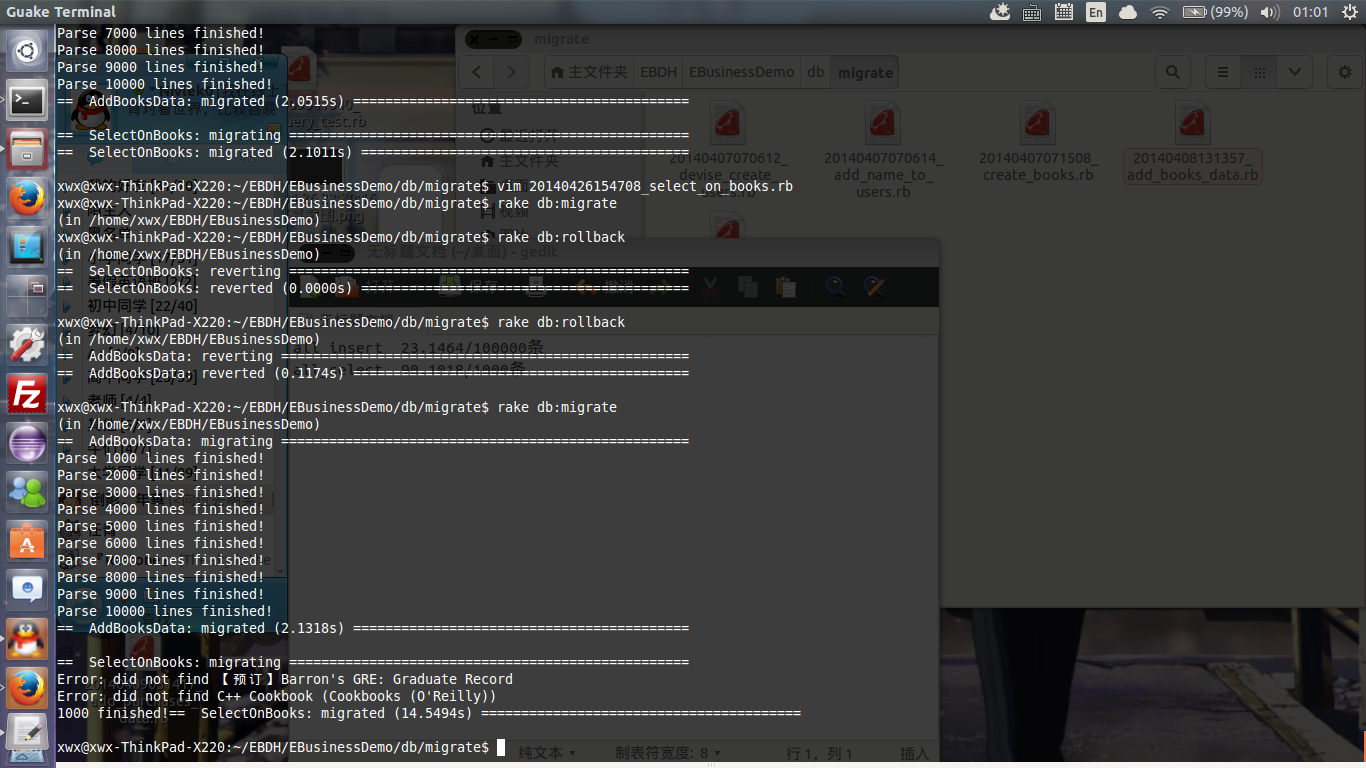
|  |  |  |  |
| --- | --- | --- | --- |
|  | 1000条记录/块 | 3000条记录/块 | 10000条记录/块 |
| Map时间(s) | 3 | 5 | 3 |
| Reduce时间(s) | 17 | 11 | 13 |



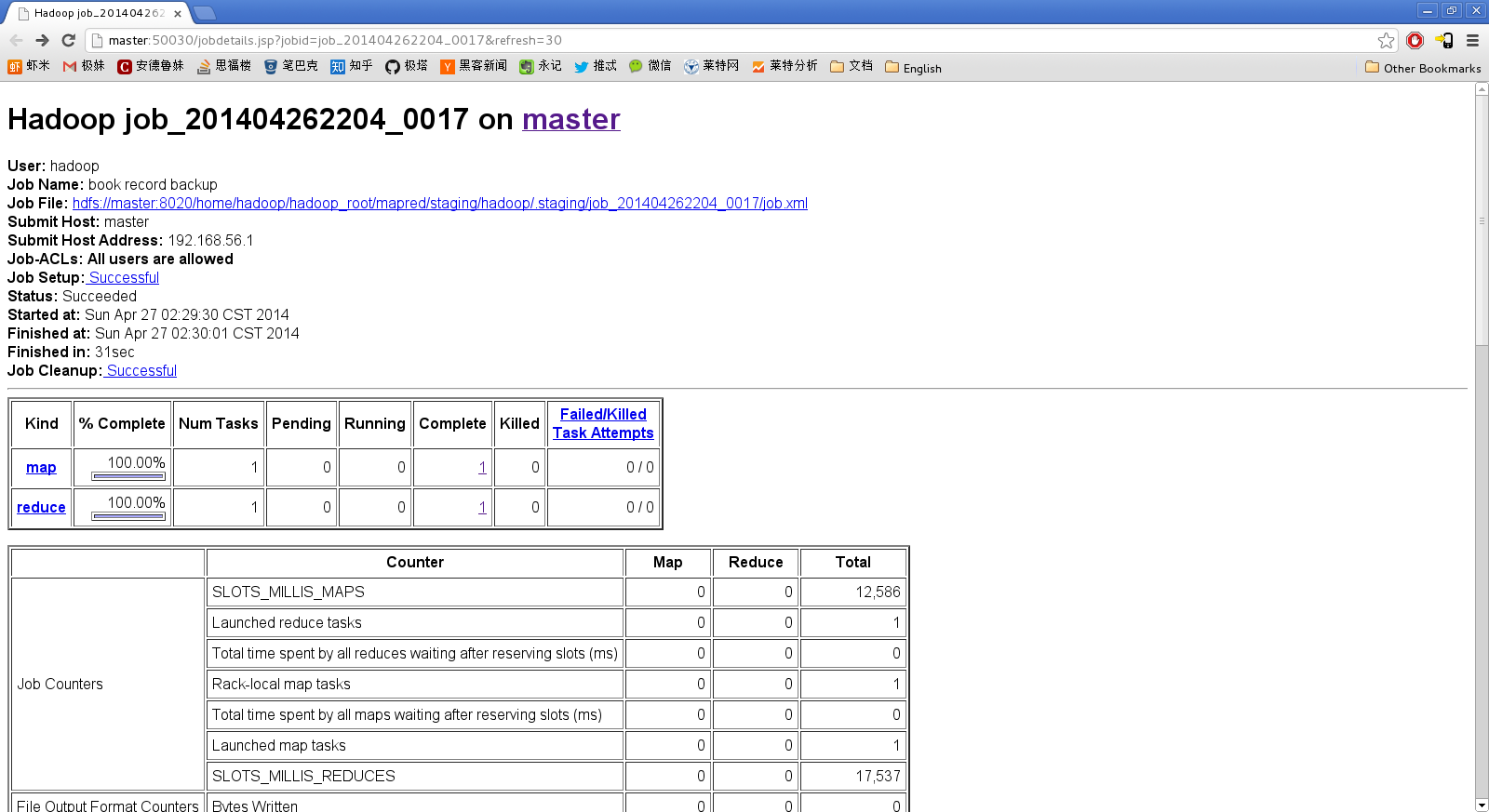
插入所有数据所花费的时间 23.1464s



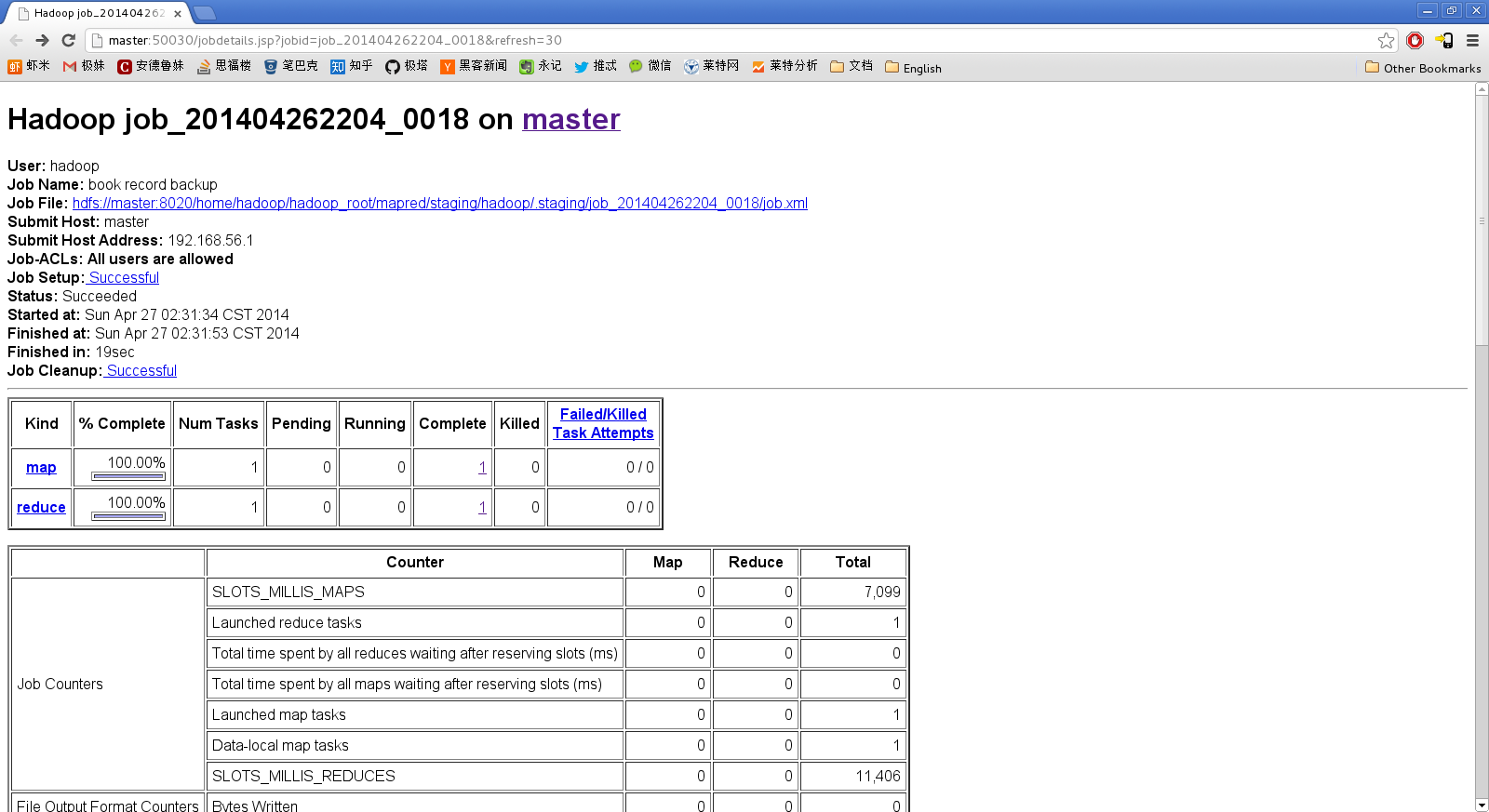
查询1,000条数据所花费时间



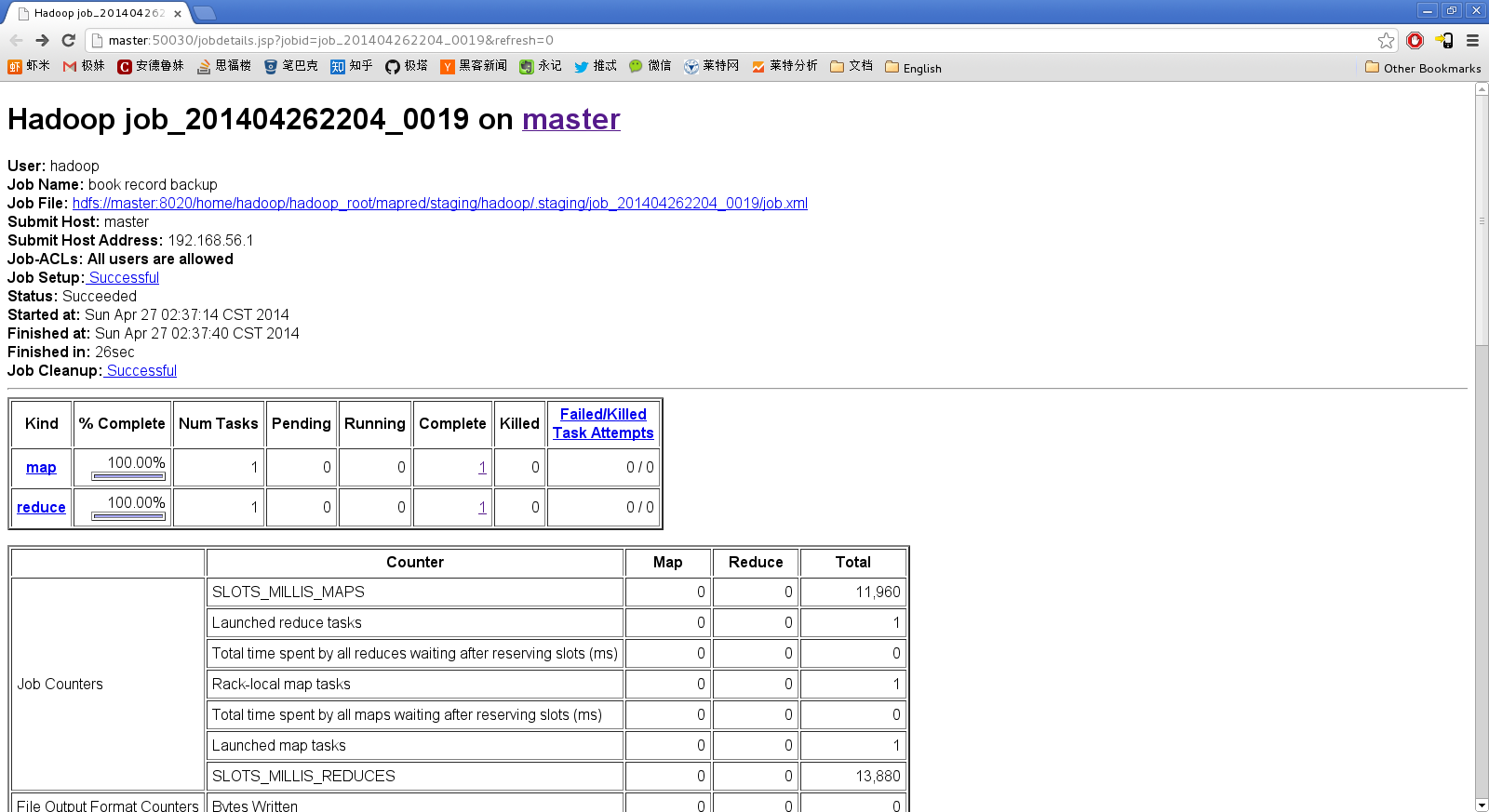
在本地插入10,000条数据 与 查询1,000条数据所花费时间



按照1000条记录/块上传到Hadoop集群的结果



按照3000条记录/块上传到Hadoop集群的结果



按照10000条记录/块上传到Hadoop集群的结果

MySQL本地插入代码：

|  |
| --- |
| require 'csv'  class SelectOnBooks < ActiveRecord::Migration  def up  counter = 0  real = 0  conn = ActiveRecord::Base.connection  i = 0  CSV.foreach("/tmp/small\_size.csv") do |items|  begin  if (counter % 10 == 0)  item = items.map{|items| Book.sanitize(item)}  name = items[0]  # print "name: #{name} \n"  begin  find = Book.find(:all, :conditions=>"title = '#{name}'" )  rescue  print "Error: did not find #{name} \n"  end  real = real + 1  if (real % 1000 == 0)  print "#{real} finished!"  end  end  counter = counter + 1  rescue  end  #print find  end  end  def down  end  end |

Hadoop集群插入实验代码：

|  |
| --- |
| package com.lightxu;    import java.io.\*;  import java.util.\*;  import java.net.URI;    import org.apache.hadoop.fs.\*;  import org.apache.hadoop.conf.\*;  import org.apache.hadoop.io.\*;  import org.apache.hadoop.util.\*;  import org.apache.hadoop.mapreduce.\*;  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  import org.apache.hadoop.util.GenericOptionsParser;    public class BookRecordBackup {  public final static int blockSize = 10000;    public static class Map extends Mapper<LongWritable, Text, LongWritable, Text> {  private Text word = new Text();    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  String line = value.toString();  LongWritable block\_id = new LongWritable(Long.parseLong(line.substring(0, line.indexOf(" "))) / blockSize);  context.write(block\_id, value);  }  }    public static class Reduce extends Reducer<LongWritable, Text, NullWritable, NullWritable> {  public void reduce(LongWritable key, Iterable<Text> values, Context context)  throws IOException, InterruptedException {  Configuration conf = context.getConfiguration();  String file\_prefix = conf.get("backup.file.prefix");  String file\_path = file\_prefix + "/" + key.toString();  FileSystem fs = FileSystem.get(URI.create(file\_path), conf);  OutputStream out = fs.create(new Path(file\_path));  for (Text value : values) {  out.write(value.toString().getBytes("UTF-8"));  }  out.flush();  out.close();  context.write(null, null);  }  }    public static void main(String[] args) throws Exception {  Configuration conf = new Configuration();  String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();  if (otherArgs.length != 2) {  System.err.println("Usage: BookRecordBackup <in> <out>");  System.exit(2);  }  conf.set("backup.file.prefix", otherArgs[1]);    Job job = new Job(conf, "book record backup");  job.setJarByClass(BookRecordBackup.class);    job.setMapperClass(Map.class);  job.setReducerClass(Reduce.class);    job.setInputFormatClass(TextInputFormat.class);  job.setMapOutputKeyClass(LongWritable.class);  job.setMapOutputValueClass(Text.class);  job.setOutputKeyClass(NullWritable.class);  job.setOutputValueClass(NullWritable.class);    FileInputFormat.addInputPath(job, new Path(otherArgs[0]));  FileOutputFormat.setOutputPath(job, new Path(otherArgs[1]));    System.exit(job.waitForCompletion(true) ? 0 : 1);  }    } |

1. **实验分析：**

由上述的实验结果，我们可以得出：在电商系统的商业需求下，单条记录的查询速度，使用hadoop分布式存储，将比使用单一的传统数据库效率要高的多。虽然在存储数据方面把所有数据上载到Hadoop集群耗费了与Mysql插入相当的时间（17秒vs 23秒），但由于Hadoop在启动作业时有很多额外开销，当数据量更大以后其优势将会显现的越明显。而当数据转存到HDFS后，Mysql的查询就变得快速了许多。(0.002秒 vs 0.09秒)

另外，我们发现在利用Map Reduce存储的时候，分块的大小对存储速度也颇有影响，并且不是呈线性关系，例如在本实验的三种情况中，选取3000条记录/块是最佳选择，这一参数同时影响到了Map和Reduce两方面。