DESIGN EXPERIMENT OF OPEN-STREAMER LIBRARY

目录

一 实验设计说明

Open-Streamer[1]提供了非常丰富的流处理算法，包括线性分类器，回归器，以及有关计数的算法，本实验着重对基数计数算法进行各项指标的测试，包括在 Storm 平台上该算法的吞吐能力，以及不同参数对同一个算法的正确率的影响，以及同一个参数下不同数据量对正确率的影响，以及不同算法的正确率的比较等，细节部分将在实验过程中给出。

该实验首先实验证明基数计数算法对计算机的资源占用率不具有增量特点，随着时间推移，无论数据量如何庞大，算法使用的资源都保持在一定范围之内，主要指标为内存使用率和 CPU 使用率，这也是为什么基数估计算法可以部署在 Storm 平台之上的一大原因，再多的数据量也不会拖垮整个 Storm 集群而影响运行在 Storm 集群之上的其他业务。

接着，将对基数估计算法的测试进行设计与实验，以多次实验取平均的思想进行实验数据的获取，然后借助可视化工具将结果展示出来，以便可以快速方便的获取实验结果。

实验最后将就本次实验，总结实验过程中遇到的问题与解决这些问题的方法，然后总结 Open-Stream 各个算法类型的典型使用场景。

二、实验环境

一、环境介绍

本实验将基于 Open-streamer 库，编写实验代码，然后运行在 Storm 集群之上，下面介绍实验设计过程中的主要环境。

（1）、实验开发语言：Java
（2）、JDK 版本：Java 1.8.0_121
（3）、开发 IDE: IntelliJ IDEA 2016.3
（4）、Storm 版本：Storm 0.9.7
（5）、Zookeeper 版本：Zookeeper 3.4.5
（6）、操作系统：Ubuntu 14.0.4

二、环境搭建

环境搭建主要是 Storm 集群的搭建，其他环境比如 Java 环境搭建较为简单不再赘述，项目用 Maven 来解决依赖问题。

（1）、首先准备 3 台虚拟机，修改 IP 分别为 192.168.1.1,192.168.1.2,192.168.1.3
（2）、三台虚拟机的配置都是单核 2399MHZ CPU ， 2Gb 内存
（3）、为三台虚拟机取个别名，直接在文件 /etc/hosts 里面增加如下内容就可以：
192.168.1.1 master

[1] https://github.com/pandening/open-streamer/

192.168.1.2 slave1

192.168.1.3 slave2

注意要在三台虚拟机上都要做上面的操作，否则我们在用别名来取代 IP 之后将会找不到对应的主机。

（4）、下载 Zookeeper 和 Storm，可以在下面的地址下载：

Zookeeper：http://zookeeper.apache.org/

Storm：http://storm.apache.org/

注意下载的版本，不同的版本具有不一样的特性，可能并不适合本实验。

（5）、安装 Zookeeper

如果你觉得安装 Zookeeper 集群麻烦了一些，那么你完全可以选一台做 zook 服务器就可以了，当然，为了安全和效率考虑，我将使用三台虚拟机搭建一个 Zookeeper 集群。关于配置文件，只需要配置一次，然后复制到其他机器上就可以了，可以使用 linux 下的 scp 命令来做这件事情。

为了最简化 Zookeeper 配置，您只需要将 Zookeeper 目录下的 zoo_sample.cfg 文件改名为 zoo.cfg，然后添加如下内容告诉当前机器其他 Zookeeper 机器的地址和端口就可以了，对于该实验来说，可以添加如下内容到 zoo.cfg 文件：

server.1 = 192.168.1.1:2888:3888

server.2 = 192.168.1.2:2888:3888

server.3 = 192.168.1.3:2888:3888

因为已经为每台机器取了别名，所以可以使用下面的配置来简化配置：

server.1 = master:2888:3888

server.2 = slave1:2888:3888

server.3 = slave2:2888:3888

上面的配置格式为:server.No = host:port1:port2，server 之后的 No 代表 Zookeeper 集群的机器号，host 可以是别名，也可以是 IP 地址，用来找到机器，之后的第一个端口号用来从 follower 机器连接到 leader 机器，第二个端口号用来进行 leader 选举。

配置好后分发到各个机器上就可以了，下面就可以测试 Zookeeper 是否配置成功。

在每一台机器上，以 root 身份运行如下命令：

1、首先到 Zookeeper 的 bin 目录，或者，也可以将此目录增加在环境变量中，那样就不需要到 Zookeeper 目录下就可以直接运行命令了，关于如何添加环境变量，可以参考相关资料。

2、运行命令 zkServer.sh start

在每一台机器上都运行了上面的命令之后，我们可以在每一台机器上运行命令：

zkServer.sh status 来查看结果，如果显示为 leader 或者 follower，那就代表 Zookeeper 集群已经在运行了。

（6）、安装 Storm 集群

对于安装 Storm，你必须要做的就是：

1、 告诉 Storm 集群 Zookeeper 集群的位置，因为 Storm 集群依赖 Zookeeper 集

群来协调工作，所以这是必须的，你需要到 Storm 的 conf 目录下面，打开文件 Storm.yaml，然后添加如下代码：

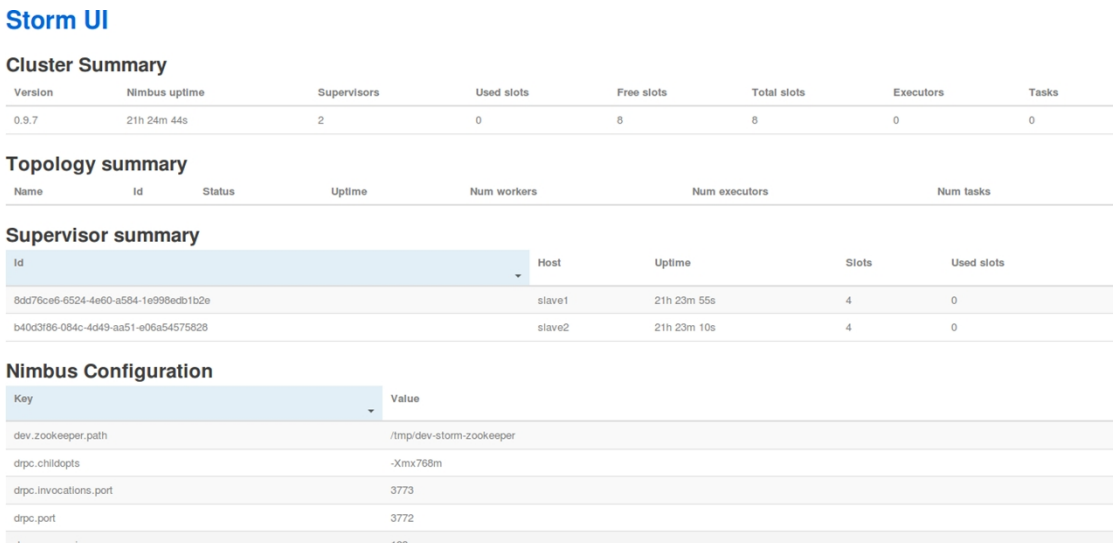
storm.zookeeper.servers:

- “192.168.1.1”
- “192.168.1.2”
- “192.168.1.3”

2、 然后你需要指定一个节点为 nimbus 节点，Storm 具有两种类型的节点，nimbus 个 Supervisor，nimbus 节点是主控节点，用来资源分配以及对 Topology 的任务调度，supervisor 则是正真的工作节点，负责从 nimbus 那里获取任务，然后启动 Worker 来执行任务，完成后将结果返回。下面的配置信息就是指定了节点：192.168.1.1 为 nimbus 节点：
Nimbus.host :”192.168.1.1”

在做了上述的配置之后，将配置文件分发到其他的机器上面，然后就可以通过下面的步骤来检验 Storm 集群是否安装成功（Zookeeper 集群已经启动的情况下）：

1、 定为到 Storm 文件下的 bin 目录
2、 在 nimbus 节点上面运行：storm nimbus ，在 supervisor 节点上运行: storm supervisor
3、 在 nimbus 节点上运行命令：storm ui，然后我们就可以通过浏览器用地址：http://127.0.0.1:8080/来查看集群的信息，如果出现如下的画面，说明 Storm 集群安装成功了。



图表 1Storm UI

三、数据

一、需求分析

对于基数计数算法，我们要首要验证的就是算法的正确性，因为该算法库里面的计数估计算法都是概率算法，因为在大数据环境下，不可能将所有数据都存储起来，只能损失一部分正确性来做估计，所以该算法库下的基数计数算法并不适合于那种要求绝对精确的场景。利用 java 的 Random 对象可以产生各种类型的随机数，为了减少生成数据的重复性，使用 double 类型的随机数生成器，这样在长生如几千万数据量的需求下，几乎不可能有重复的随机数，所以，可以做假设，用随机数生成器来生成 10000000 个 double 类型的随机数，可以假设基数就是 10000000。

二、数据集生成

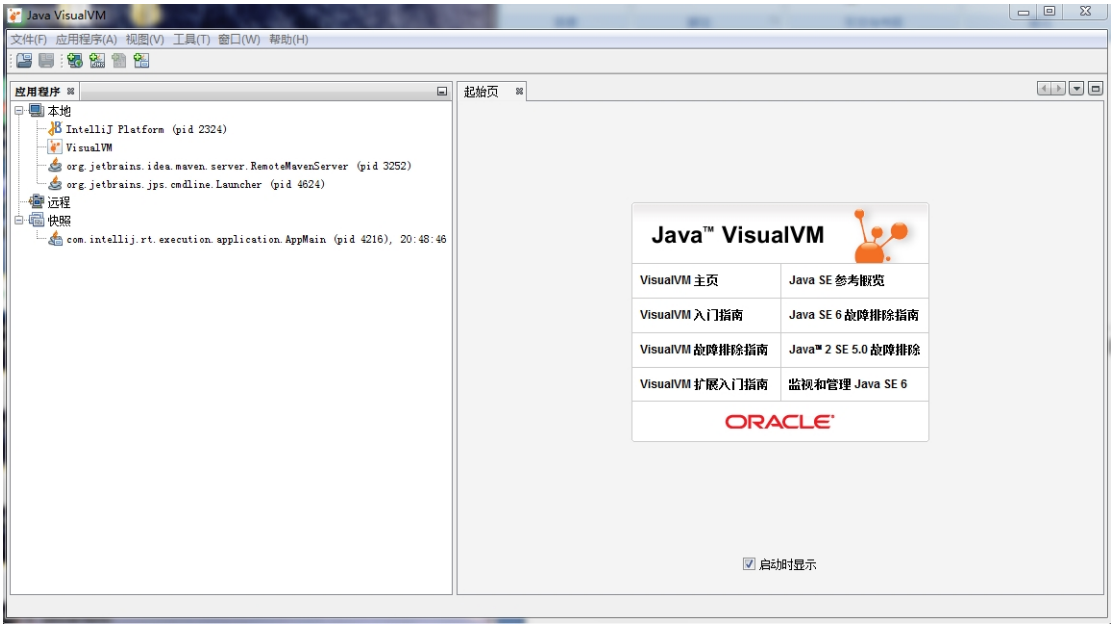
数据生成就使用随机数法，而且不做存储，模拟流式数据生成，然后让 Storm 运行基数计数算法来估计目前的基数数量。

四、实验过程

一、资源使用率测试

一、工具介绍

为了测试 java 程序在运行时的资源使用情况，可以使用多种工具，一种使用简单，功能强大的工具可以在 JDK 的安装目录下呗发现，Jconsole，或者它的升级版 jvisualvm，双击即可打开，可以看到如下界面：



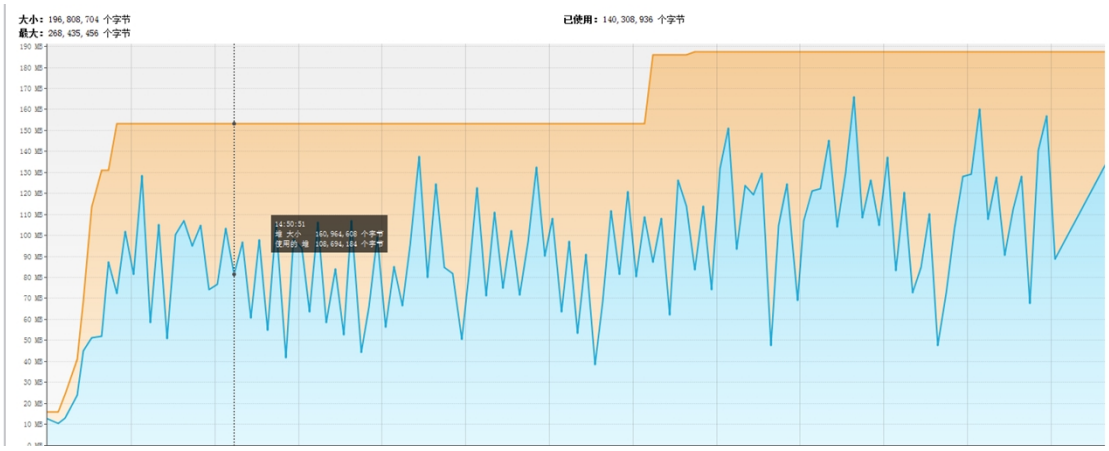
图表 2 java 资源监控工具 Jvisualvm

可以在左侧发现正在运行着的 java 程序，该工具提供本地和远程模式，可以使用后该工具监控远程运行的 java 程序的资源使用情况，对于本次实验，使用本地模式即可，点击我们需要查看的 java 进程即可看到该进程的各种资源使用情况，可以看到点击之后有四个分页，第一页为“概述”，可以看到该进程的进程号，主机，java 代码的主类，以及一些 JVM 的参数信息。第二页为“监视”，可以看到该进程的 CPU，类加载，内存使用，线程等详细信息，第三页为“线程”，可以看到该进程的所有进程的活动情况，第四页为“抽样器”，可以对机器的 CPU 以及内存进行抽 样，在本次实验中，只需要使用第二页的监视功能，就可以查看算法运行时对 cpu 以及内存的使用情况。

在了解了如何使用工具来查看 java 程序的资源使用率之后，就可以设计简单的实验来测试了，方法是，将各个算法分别运行在启动了 jvisualvm 工具的机器上面，然后点击相应的进程来查看资源的使用情况。下列测试都在 1000000 个数据量。

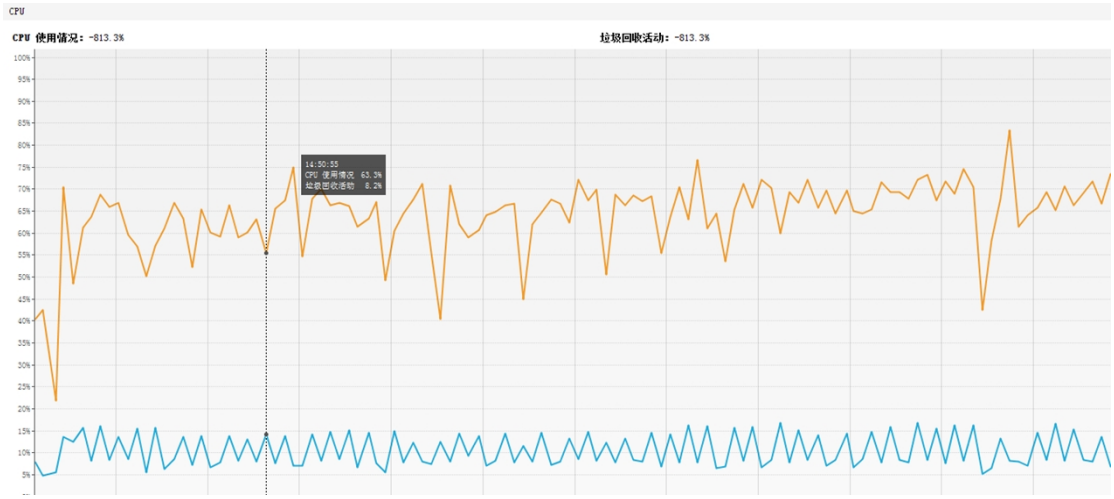
二、LogLog 算法的资源使用率

内存使用情况：



图表 3 LogLog 算法内存使用情况

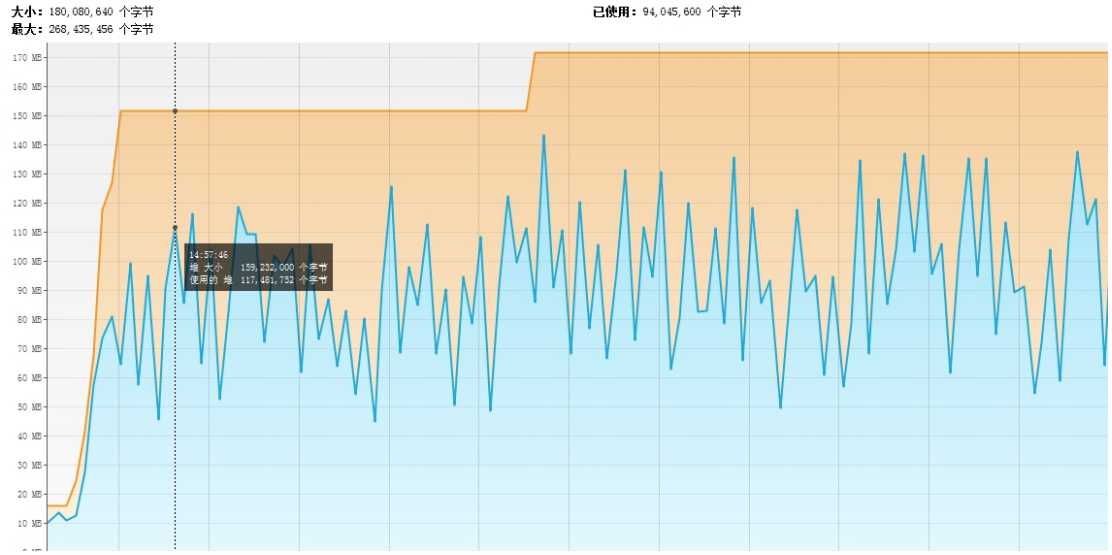
CPU 使用情况：



图表 4 LogLog 算法 CPU 使用情况

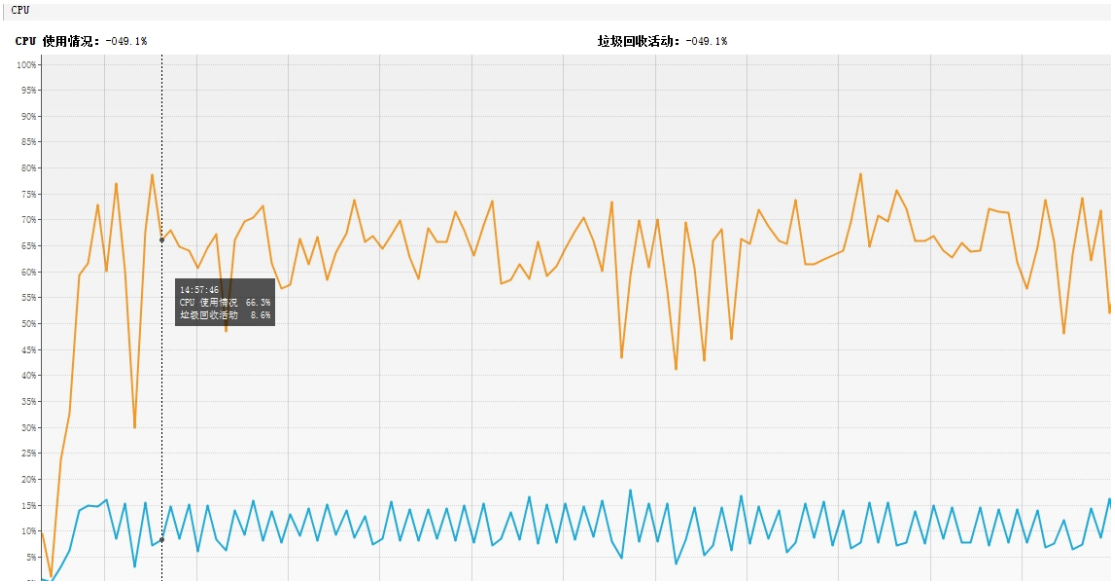
三、HyperLog 算法的资源使用率

内存使用情况：



图表 5 Hyper-Log 算法内存使用情况

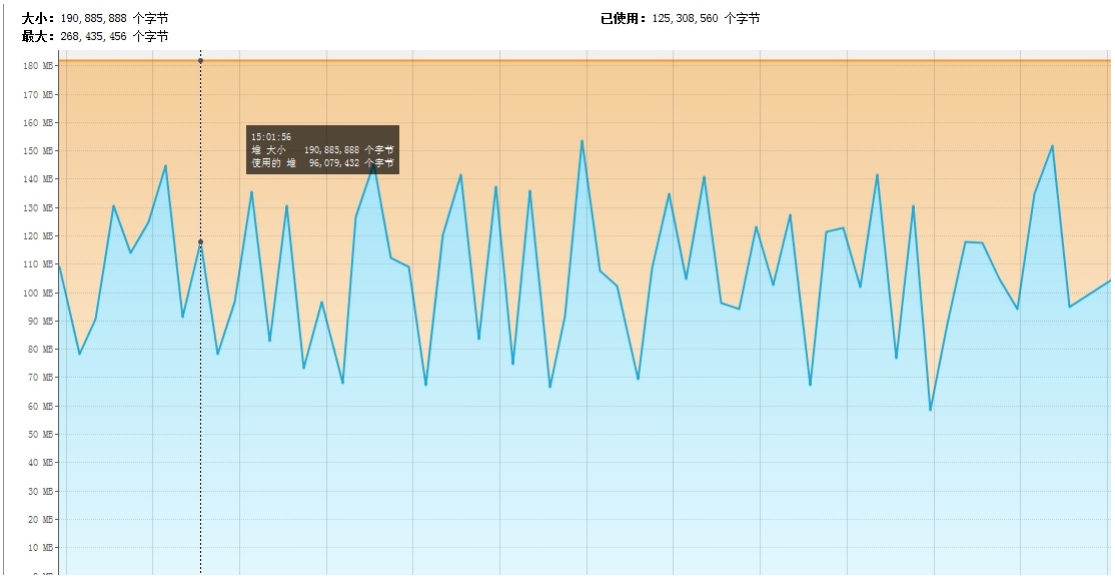
CPU 使用情况：



图表 6 Hyper-Log 算法 CPU 使用情况

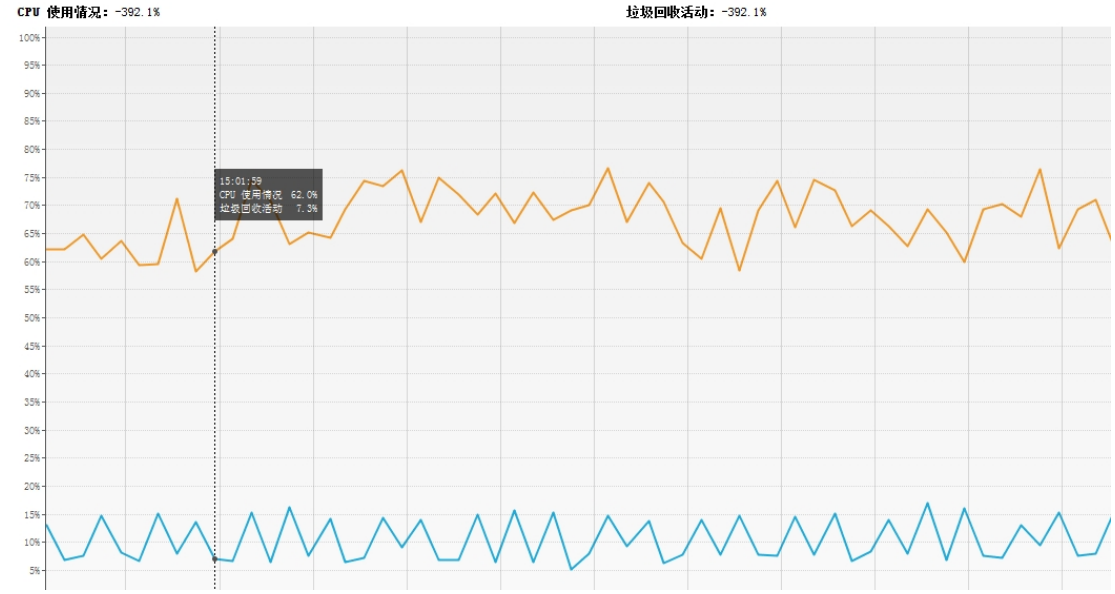
四、Linear Counting 算法的资源使用率

内存使用情况：



图表 7 Linear Counting 算法内存使用情况

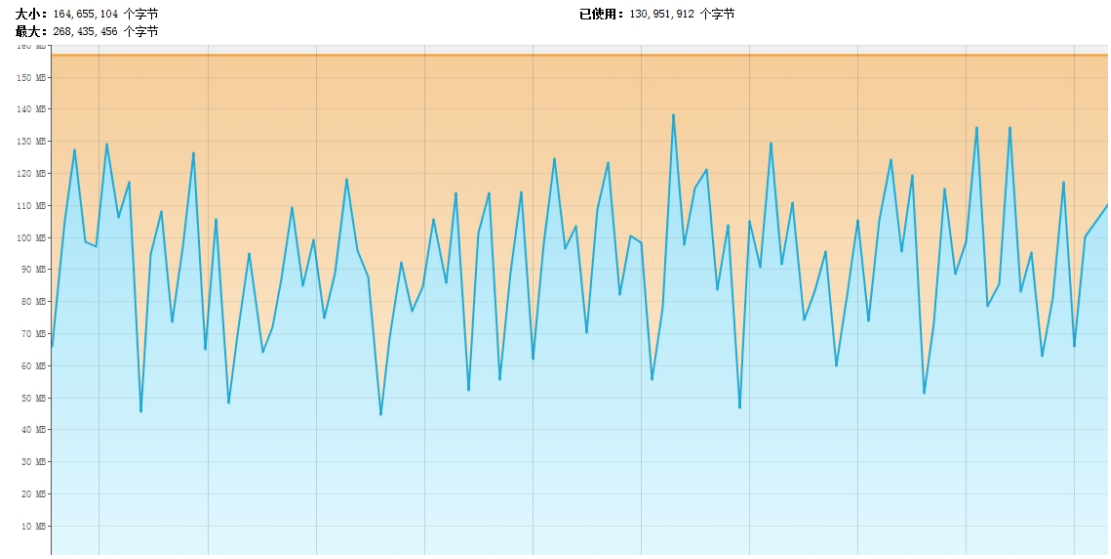
CPU 使用情况：



图表 8 LinearCounting 算法 CPU 使用情况

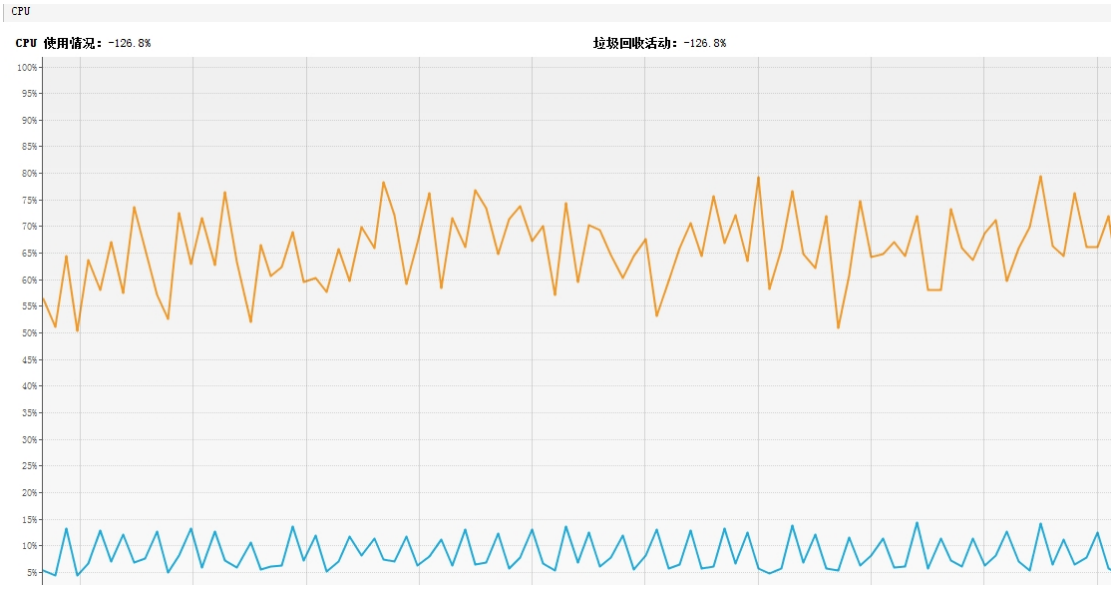
五、Adaptive Counting 算法的资源使用率

内存使用情况：



图表 9 Adaptive Counting 算法内存使用情况

CPU 使用情况：



图表 10 Adaptive Counting 算法 CPU 使用情况

二、吞吐量测试

实验设计

所谓吞吐量测试，就是测试该算法部署在 Storm 平台上之后可以处理的数据的速率，比如 10000 条/s。该部分会选择对所有算法进行测试，测试多次之后取平均值来作为实验结果。测试的数据量为 10000000，速度取样点为每隔 100000 条数据，取样率为 1/100，所以最终将会得到 100 个点，第 i 个点代表处理了（i*100000）条数据所花费的时间。

为了将算法提交到 Storm 集群上执行，需要编写符合 Storm 框架的 java 代码，然后打包成 jar 文件，然后上传到 Storm 平台上执行。

实验

下面是详细的实验步骤：

（1）、 编写测试代码，调用 open-streamer 库里面的相关接口来更新模型和做相应的查询，然后打包成 jar。对于 IDEA 工具来说，将 java 代码打包成 jar 包非常简便，可以通过下面的步骤来实现

点击 file->project structure->artifacts，然后点击那个绿色的加号，点击 jar-> modules with dependencies,然后点击 ok 按钮，回到主界面，点击 build 按钮，然后点击 build artifact 按钮，选择想要打包的 module，点击 build/rebuild 按钮就可以生成该 module 的 jar 包，找到你设置好的输出目录，就可以找到生成的 jar 包。

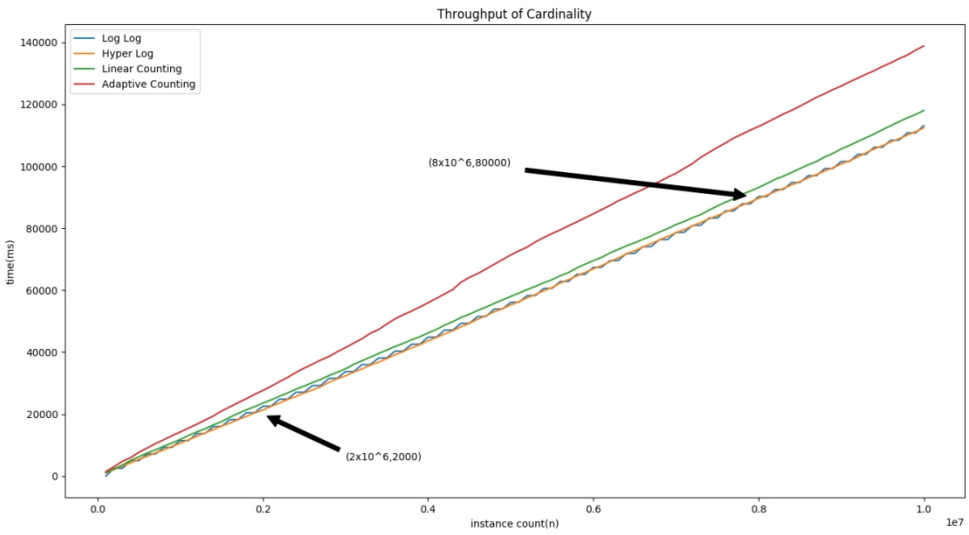
（2）、提交 jar 包到 Storm 集群，将打包好的 java 项目，通过下面的命令可以上传到 Storm 集群。

Storm jar [jar 包的文件名] [java 主类的包地址，写到类名]

（3）、提交成功之后，可以到 http://127.0.0.1:8080/去查看刚才提交的 topology，为了获得输出结果，可以去 supervisor 节点上的 Storm 日志文件目录下找类似于 worker-xxx.log 的文件，这就是 Storm 的 Worker 产生的输出日志，如果一个 supervisor 上有多个 worker 运行，就会有多个日志文件，打开之后可以看到 Storm 运行过程中的所有输出。

（4）、对每个算法进行 10 次测试之后，将取到的 100x10 个点进行求平均值处理，然后借助可视化工具来将数据可视化，一种简单的方法是使用 python 的 matplotlib 库，可以快速的将数据可视化。

下面是对每个算法进行 10 次试验之后取均值后的实验结果：



图表 11 基数估计算法吞吐量测试

横坐标为处理掉的数据量，纵坐标为使用的时间，单位是毫秒，可以看到，除了 Adaptive Counting 算法之外，其他算法所花的时间差不多是一样的，吞吐量可以达到大约 10 万条/

秒。当前的实验环境配置较低，可以达到这样的吞吐量已经很乐观了，以这样的吞吐量计算，每天可以处理的数据量大约为 80 亿条。

六、不同参数对准确率的影响测试

实验设计

该实验对 Adaptive Counting 和 LogLog 算法进行不同参数对结果的影响的测试，Adaptive Counting 算法是将 LogLog 算法与 Linear Counting 算法的结合体，需要提供一个参数（k），本实验的目的是对不同的输入 k，是否对算法的准确率有影响，本实验取了三种 k 值，分别是 5,15,25，对同一个参数进行 10 次实验，取均值代表该 k 值下的准确率数据，最后将三个取值下的数据在同一个坐标里面可视化。对于 LogLog 算法，取 k 值为：5,7,8,9,10,11，经过相同的处理之后在同一个坐标里面显示。取样率为 1/1000。

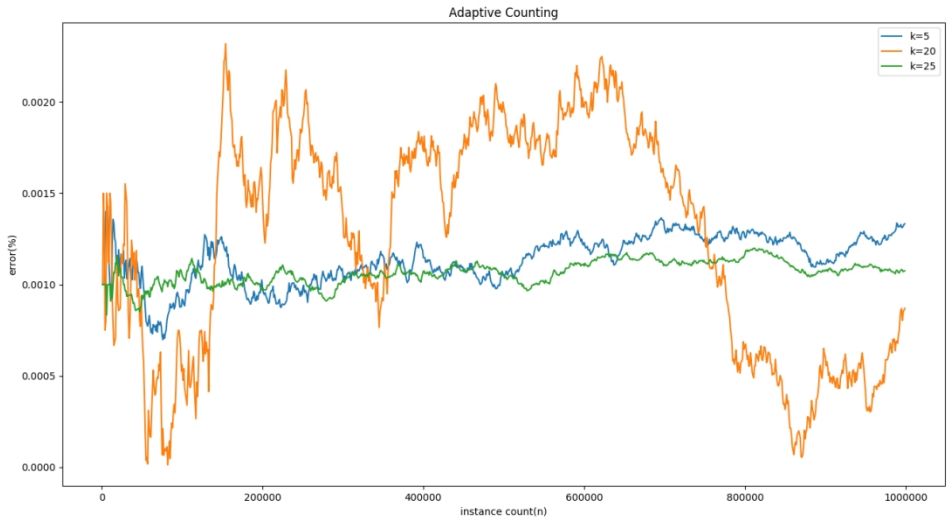
实验

下面是详细的实验步骤：

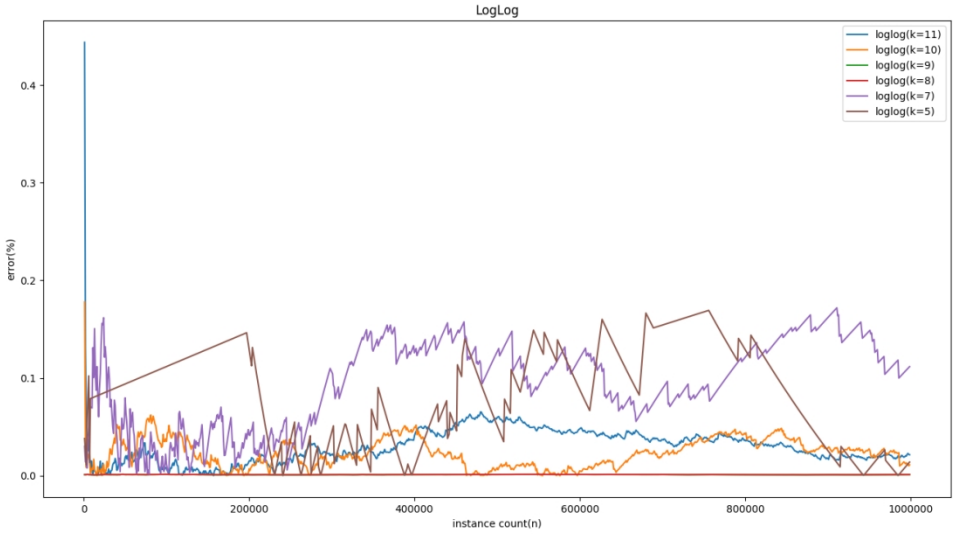
（1）、编写可以运行在 Storm 集群之上的 java 代码，算法本身应该作为参数传入测试框架，所以测试程序应该对该类算法具有通用性，这就需要所有的算法应该都实现相同的方法，这样我们可以将算法抽象成一个接口，然后作为参数传递给测试类，就可以对不同的算法进行测试。

（2）、提交 Storm 运行之后，取均值然后进行作图处理。

下面是 Adaptive Counting 算法的不同输入参数对算法的正确率影响的实验结果。



图表 12 不同参数对 Adaptive Counting 算法准确率的影响



图表 13 不同 k 值对 LogLog 算法的准确率的影响

横坐标为处理的数据量，总的数据量为 1000000 条，纵坐标为错误率，错误率的计算如下：

$\frac{|\text{predict} - \text{correct}|}{correct} * 100\ \%$,predict 为 Adaptive Counting 给出的预测值，correct 为正确值。

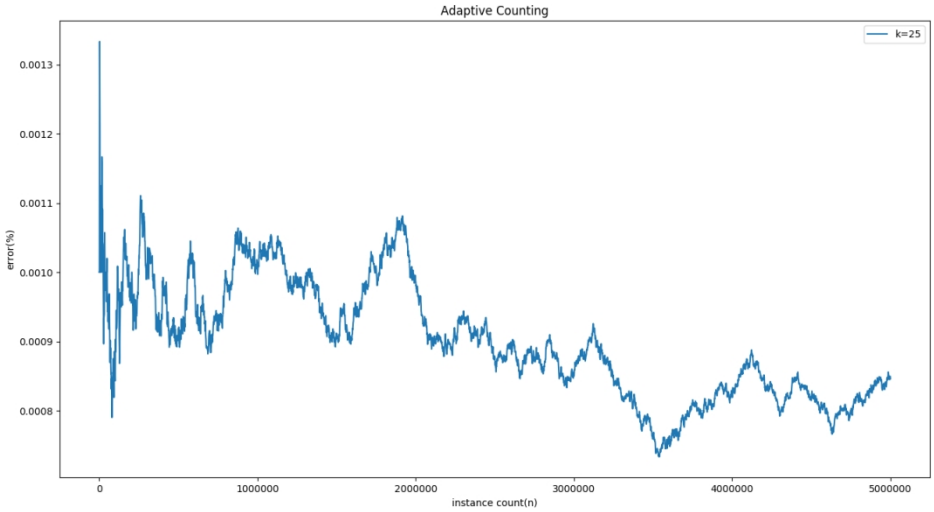
七、不同数据量对准确率的影响测试

实验设计

本实验的目的为了验证不同的数据量对算法的表现的影响，本实验依然选用 Adaptive Counting 算法来进行实验，将数据量从 1000000 增加到 5000000 条，然后去 k 为 25，运行 10 次之后取平均值，然后展示出来。取样率为 1/1000。

实验

该实验较为简单，只需要将数据量增加到 5000000 条，然后运行 10 次取平均值，下面是实验结果。



图表 14 不同数据量对 Adaptive Counting 算法的影响

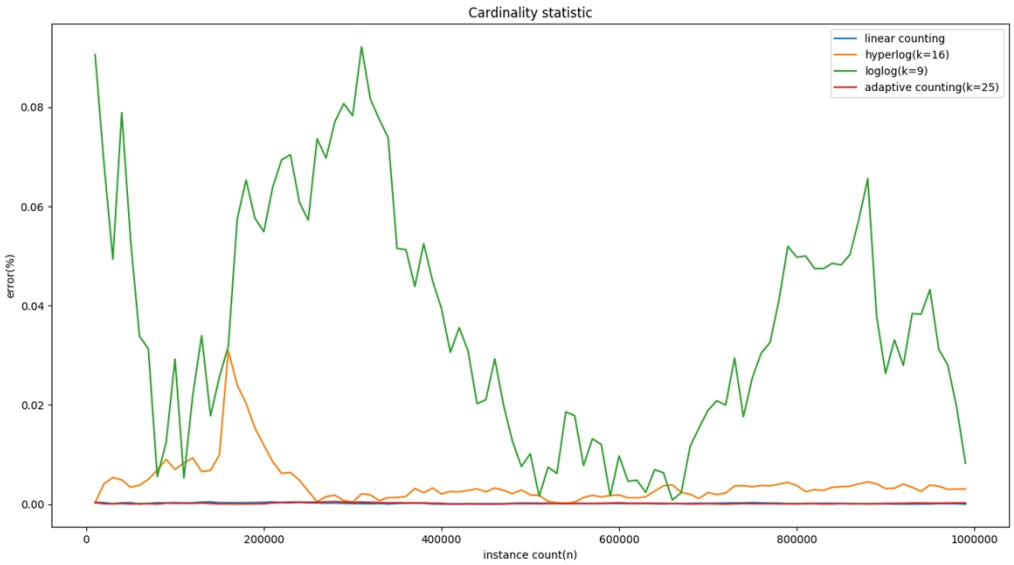
八、不同算法的准确率的对比测试

实验设计

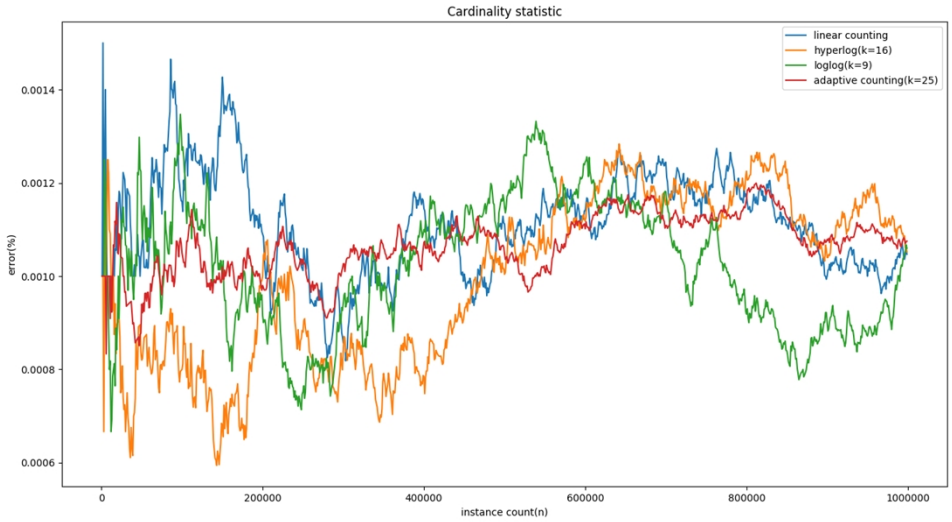
本实验的目的是比较不同基数估计算法的正确率，所以对所有算法均有实验，参与实验的算法包括 LogLog 算法，Hyper-Log 算法，Linear Counting 算法，Adaptive Counting 算法，数据量为 1000000 条数据，每个算法的结果均是由运行 10 次后取到的均值。

实验

为了减少取样的随机性，取样率分为 1/1000 和 1/10000 两种，下面是不同取样率的实验结果。



图表 15 取样率为 1/10000 的不同算法的准确率



图表 16 取样率为 1/1000 的不同算法的准确率

九、实验结果

一、资源使用测试实验

在实验结果中，可以发现，各个实验的结果均一致，无论是内存还是 CPU 占用都保持在一定的范围内，这是可以将算法移植到 Storm 这样的大数据平台的关键，因为像 Storm 这样的流处理平台，它假设启动之后就不会中断，会一直运行下去，如果某个算法的资源使用率一直在上升，那总会在一个时间节点上，集群的资源会被耗尽，整个集群将会瘫痪。次实验证明了 open-streamer 库所实现的基数计数算法均可以在 Storm 平台上永恒的运行下

去。

二、吞吐量测试实验

吞吐量测试的实验结果表明，Storm 的吞吐能力不如批处理平台 Hadoop，主要的时间花费在数据的传输上，但是对于流处理系统来说，这样的吞吐量已经可以达到大多数实时场景的需求，为了提升处理速度，可以增加 Storm 节点，升级节点的配置，可以获得更大的吞吐能力。

这也从一个侧面说明了不同的场景需要使用不同的平台来做处理，对于需要较大吞吐能力而又不要求实时性的场景下，选择 hadoop 是最合适的，对于要求实时性的场景下，就需要一个配置良好的 Storm 集群来解决问题了。流处理的使用场景具有一些固定的特点，比如实时，流数据，永恒运行等，而批处理的使用场景一般伴随着庞大的日志文件。

三、不同参数对算法影响的测试的实验

不同的参数对算法的准确率确实会有影响，从该实验结果中可以看出，k 值的选择并没有什么规律可循，需要结合多次测试结果，与数据量的大小来选择合适的 k 值，k 值不一定越大越好，也不是越小越好。

四、不同数据量对算法的影响的测试实验

该实验可以证明，对于估计算法而言，数据量越大，估计的准确率就越好，这可以理解为，数据量越多，基数估计算法的模型越有经验，估计的准确度也就越高。

五、不同算法的正确率测试

该实验对比了不同的基数估计算法的准确性，并且在不同的取样率下作了比较，可以看出，取样率越大，可以看到的细节越多，Hyper-Log 算法的表现在各个数据量下都较好，Linear Counting 算法有一定的启动时间，所以开始阶段正确率较低，但是随着数据量的增加，该算法的正确率逐渐升高，LogLog 算法的正确率在数据量变多时变好，Linear Counting 算法在数据量变大时正确率降低，而 Adaptive Counting 算法是对 Linear Counting 算法和 LogLog 算法的结合，所以 Adaptive Counting 算法的稳定性较好。

所以，在数据量较小的时候，可以使用 Linear Counting 算法来做基数估计，数据量较大时可以使用 LogLog 算法，如果不确定数据量的情况下，使用 Adaptive Counting 算法是一个比较合理的选择。

十、实验总结

经过一系列的实验，对 open-streamer 算法库的基数估计算法进行了较为详细的对比分析，通过实验得到一系列的结果。下面将列出在实验过程中遇到的问题与解决问题的方法。

（1）、Storm 安装问题，安装成功后，开启之后与安装的版本无法对应

该问题的原因是在该机器上原来已经安装过 Storm，并且没有将环境变量删掉，所以需要将原来的 Storm 的环境变量改为新安装的 Storm，或者可以直接定位到 Storm 安装目录下的 bin 目录下，然后使用 python storm ..来启动 Storm 集群，这样就可以解决。

（2）、打包 Java 代码后运行时出现错误，提示某个文件多余

该问题的原因是，项目使用 maven 来解决依赖问题，一些依赖会冲突，也有可能项目的依赖于 Storm 环境所提供的资源冲突，所以，一种解决方法是找出依赖冲突的地方，去掉依赖冲突项，或者，可以在打包之后打开 jar 包，然后将提示冲突或者多余的文件删掉就可以了。

（3）、Storm 集群启动之后，在 UI 里面看不到相应的 Supervisor，或者错误显示 Supervisor 数量。

该问题出现的原因是，Storm UI 获取信息一部分是通过扫描日志来显示的，所以可能某些原来遗留下来的日志文件会影响显示结果，解决方法是删除那些初始时不存在的文件，然后重新开启 Storm，就可以看到正确的显示结果。

（4）、在 Storm 的 nimbus 节点上提交了 Topology 之后，在 UI 里面一直看不到该 Topology 运行的痕迹

该问题出现的原因是，Spout 设置的不合理，或者提交的 Topology 本来就没有数据流动，解决方法是查看代码，如果本来就不存在数据流动的代码，那么属于正常，如果设置了 Spout 并且有数据被 emit，那么就要看看是否设置的并行度过高，过高的并行度将有可能拖垮整个 Storm 集群。应该设置合理的组件并行度。

（5）、在 Storm 集群上提交了 Topology 之后，Topology 没有运行

该问题出现的可能情况为，nimbus 无法获取足够的资源给该 topology 运行。Nimbus 节点主要负责资源分配和任务调度，当没有可用的资源，比如没有空闲的 Slot 时，storm 的 nimbus 节点将会把新提交的 Topology 挂起，等其他 Topology 被删掉之后空余出来资源之后再重新分配资源给挂起的 Topology。所以解决方法是，删掉无用或者测试的 Topology。