在云平台安装 Spark 并运行 KMeans

> 李易非 3160105705
>
> 熊苗 3160104714
>
> 李灿晨 3160105187

之前的实验中我们尝试了 Hadoop 的配置以及使用，并使用 Hadoop 完成了包括 WordCount、KMeans 以及 NaiveBayes 在内的多种算法。然而，虽然 Hadoop 的分布式特性能够将计算负载分配在多个节点上以减少对单个节点的压力以及运算时间，但是，根据 Hadoop MapReduce 的特性，对于需要迭代进行多轮 MapReduce 的程序来说，在不同的 MapReduce 过程之间传递数据是十分困难的，无论是通过包括 HBase 在内的数据库读写还是直接在 HDFS 上存取，都会花费大量时间在数据的获取与释放上，而这些操作是无意义的。而 Spark 框架拥有先进的 Resilient Distributed Datasets（RDD）特性，使得每轮操作中的数据能够被分布式地缓存在不同节点的内存中，极大地减少了反复从 HDFS 读取文件所造成的时间浪费。同时，Spark 作为计算框架，与 Hadoop 生态系统中的 HDFS，HBase，Hive 等兼容，能够为 Hadoop 集群的性能带来显著的提升。

在本次试验中，我们将在集群上配置 Spark，并使用 Scala 在 Spark 框架下重写之前完成的 KMeans 算法，并将在 Spark 框架下的运行性能与在 Hadoop MapReduce 下的性能进行对比。

Spark 配置

在本部分中，我们将会在已经配置完成 Hadoop 3 的服务器上进行 Spark 框架的配置，并运行简单的测试模型以验证 Spark 的配置情况。

Scala 环境配置

通过 `apt-get install scala` 命令下载 Scala。

通过更改 `.bashrc` 文件来配置 Scala 的相关环境变量：在文件中增加如下代码：

```
export SCALA_HOME=/usr/share/scala-2.11
```

在终端中输入 `scala` 以启用 Scala Console，输入一行简单的代码以确认 Scala 安装成功。

```
[root@host0 ~]$ scala
Welcome to Scala version 2.11.6 (OpenJDK 64-Bit Server VM, Java 1.8.0_181).
Type in expressions to have them evaluated.
Type :help for more information.

scala> println("Hello, world.")
Hello, world.
```

sbt 环境配置

我们采用 `scala` 作为我们的开发语言，所以需要通过 `sbt` 进行打包，生成 `*.jar` 文件，来提交到 spark 集群进行工作。

- 本机在 sbt 官方网站 下载 `sbt-1.2.6.tgz`，并通过 `scp ~/Downloads/sbt-1.2.6.tgz root@59.101.111.30:~` 发送至 master 服务器中

```
sudo tar -zxvf sbt-1.2.6.tgz # 解压 sbt 压缩包
mv sbt /usr/local/sbt
```

- 修改 `~/.bashrc`

```
export SBT_HOME=/usr/local/sbt
export PATH=$PATH:$SBT_HOME/bin
```

- 测试 sbt 是否正常安装

```
[root@host0 ~]$ sbt
[info] Updated file /root/project/build.properties: set sbt.version to
1.2.6
[info] Loading project definition from /root/project
[info] Updating ProjectRef(uri("file:/root/project/"), "root-
build")...
[info] Done updating.
[info] Set current project to root (in build file:/root/)
[info] sbt server started at
local:///root/.sbt/1.0/server/27dc1aa3fdf4049b492d/sock
sbt:root> exit
[info] shutting down server
```

Spark 环境配置

- 下载spark压缩包
- 利用scp传送到云服务器上
- 使用`tar xvf spark-1.3.1-bin-hadoop2.6.tgz`解压提取
- 移动到相应文件夹下并设置Spark环境
  - 将`export PATH = $PATH:/usr/local/spark/bin`添加到`~/.bashrc`文件中
  - 通过`source ~/.bashrc`使命令生效
- 检验spark安装是否安装成功
  - 输入`spark-shell`
  - 出现spark字样即可认为安装配置成功

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.4.0
      /_/
```

- 修改 `~/.bashrc`

```
export SPARK_HOME=/usr/local/spark/spark-2.4.0-bin-hadoop2.7
export PATH=$PATH:$SPARK_HOME/bin
```

  - 没有将 `$SPARK_HOME/sbin` 的路径也加入 `$PATH` 中，因为会与 `$HADOOP_HOME/sbin` 下的一些命令产生冲突；
- 修改 `$SPARK_HOME/conf/spark-env.sh`
  - `cp spark-env.sh.template spark-env.sh`
  - 在 `spark-env.sh` 中填入如下内容

```
export JAVA_HOME=/usr/lib/jvm/default-java
export SCALA_HOME=/usr/share/scala-2.11
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_CONF_DIR=/usr/local/hadoop/etc/hadoop
export SPARK_MASTER_IP=host0                    #   指定 master 地址
export SPARK_WORKER_MEMORY=4g                   #   指定 worker 的内存大小
export SPARK_WORKER_CORES=2                     #   指定每个 worker 的核数
export SPARK_WORKER_INSTANCES=1                 #   每个 worker 需要创建几个 instance
```

- 修改 `$SPARK_HOME/conf/slaves`
  - `cp slaves.template slaves`
  - 在 `slaves` 中填入以下内容（要与自己在 `/etc/hosts` 中指定的别称相同
    - `slave1`
    - `slave2`
    - `slave3`
    - `slave4`
- 将修改后的配置，同步到各个 slave 节点

```
rsync -av /usr/local/spark/spark-2.4.0-bin-hadoop2.7
slave1:/usr/local/spark/spark-2.4.0-bin-hadoop2.7
rsync -av /usr/local/spark/spark-2.4.0-bin-hadoop2.7
slave2:/usr/local/spark/spark-2.4.0-bin-hadoop2.7
rsync -av /usr/local/spark/spark-2.4.0-bin-hadoop2.7
slave3:/usr/local/spark/spark-2.4.0-bin-hadoop2.7
rsync -av /usr/local/spark/spark-2.4.0-bin-hadoop2.7
slave4:/usr/local/spark/spark-2.4.0-bin-hadoop2.7
```

spark 集群环境启动

在这个部分中，我们将 spark 集群运行 hdfs 之上

- 启动 hdfs 系统 `start-dfs.sh`
- 启动 spark 集群 `$SPARK_HOME/sbin/start-all.sh`

```
starting org.apache.spark.deploy.master.Master, logging to
/usr/local/spark/spark-2.4.0-bin-hadoop2.7/logs/spark-root-
org.apache.spark.deploy.master.Master-1-host0.out
slave1: starting org.apache.spark.deploy.worker.Worker, logging to
/usr/local/spark/spark-2.4.0-bin-hadoop2.7/logs/spark-root-
org.apache.spark.deploy.worker.Worker-1-slave1.out
slave2: starting org.apache.spark.deploy.worker.Worker, logging to
/usr/local/spark/spark-2.4.0-bin-hadoop2.7/logs/spark-root-
org.apache.spark.deploy.worker.Worker-1-slave2.out
slave4: starting org.apache.spark.deploy.worker.Worker, logging to
/usr/local/spark/spark-2.4.0-bin-hadoop2.7/logs/spark-root-
org.apache.spark.deploy.worker.Worker-1-slave4.out
slave3: starting org.apache.spark.deploy.worker.Worker, logging to
/usr/local/spark/spark-2.4.0-bin-hadoop2.7/logs/spark-root-
org.apache.spark.deploy.worker.Worker-1-slave3.out
```

- 此时利用 `jps` 命令可以在 slave 和 master 结果上看到不同的结果
  - master 上 `jps`

```
6641 NameNode
6993 SecondaryNameNode
7401 Master
7465 Jps
```

  - slave 上 `jps`

```
2224 Worker
1732 DataNode
2308 Jps
```

spark-shell wordcount 示例

- 利用 `spark-shell` 我们可以得到一个交互式的环境，运行一些简单的 spark 指令，譬如如下命令可以完成一次简单的 wordcount

```
scala> val textFile = sc.textFile("/user/simple_word").cache()
scala> val wordcount = textFile.flatMap(line => line.split("
")).map(word => (word, 1)).reduceByKey(_ + _)
scala > wordCount.collect().foreach(println(_))
scala> wordcount.collect.foreach(println(_))
(Catholics,,1)
(someone,1)
(call,1)
(this,6)
(doubt,,1)
(discussion,1)
(now,,1)
(magazine,,1)
(sort,1)
(have,7)
(think,1)
(policy,2)
(plenty,1)
(some,4)
(well.,1)
(simple,1)
(This,1)
(we,1)
...
```

spark-submit wordcount 示例

我们采用 `scala` 作为我们的开发语言，所以需要通过 `sbt` 进行打包，生成 `*.jar` 文件，来提交到 spark 集群进行工作

- sbt 需要严格的目录来进行项目代码组织

```
src/
  build.sbt
  main/
    resources/
       <files to include in main jar here>
    scala/
       <main Scala sources>
    java/
       <main Java sources>
  test/
    resources
       <files to include in test jar here>
    scala/
       <test Scala sources>
```

```
    java/
       <test Java sources>
```

- `build.sbt` 中配置一些 metadata

```
name := "WordCount"

scalaVersion := "2.11.6"

libraryDependencies += "org.apache.spark" %% "spark-core" % "2.4.0"
```

- `scala` 文件夹下，放入我们的 `WordCount.scala` 即可

```
    import org.apache.spark.SparkContext
    import org.apache.spark.SparkContext._
    import org.apache.spark.SparkConf

    object WordCount {
        def main(args: Array[String]) {
            val conf = new SparkConf().setAppName("WordCount")
            val sc = new SparkContext(conf)
            val textFile = sc.textFile("/user/spark_wordcount/sample")
            val counts = textFile.flatMap(line => line.split("
")).map(word => (word, 1)).reduceByKey(_ + _)
            counts.collect.foreach(println(_))
        }
    }
```

- 进入项目根目录，`sbt pacakge`，第一次可能需要下载大量的相关文件，之后速度很快

```
[root@host0 WordCount]$ sbt package
[info] Loading project definition from /root/spark/WordCount/project
[info] Loading settings for project wordcount from build.sbt ...
[info] Set current project to wordcount (in build
file:/root/spark/WordCount/)
[success] Total time: 2 s, completed Nov 27, 2018 8:28:24 PM
```

- 打包后的文件在 `target/scala-2.11/` 下
- `spark-submit --class "WordCount" --master spark://host0:7077 target/scala-2.11/*.jar` 后即可正常运行

使用 Scala 完成适用于 Spark 的 KMeans 算法

在之前的实验中，我们使用 Java 完成了适用于 Hadoop MapReduce 框架的 KMeans 算法。由于本次试验所完成的算法主要是出于对比目的，而不是进行数据分析，我们没有对之前的算法进行优化，也不会对算法产生的输出进行实际情境下的分析，我们只是在之前算法的基础上进行重构以适应 Spark 框架的运算需要，并对结果与之前 KMeans 算法的结果进行对比，以确认我们本次完成的 KMeans 算法与之前的实现基本相同，从而达到在 Spark 与 Hadoop MapReduce 的性能对比中控制变量的目的。

我们使用伪代码来表示我们的 KMeans 算法思路：

```
input: center_vectors, data_vectors, k, max_iteration
converge = false, iteration = 0
while not converge or iteration < max_iteration:
    for vector in data_vectors:
        vector.cluster = closest_cluster(with: vector)
    for center in center_vectors:
        center.position = average(center.vectors)
output: center.index, center.vectors.count
```

基础方法实现

KMeans 算法需要一些包括距离计算，最近中心判断在内的简单函数实现以支持运行，在本部分，我们将会给出相应方法在 Scala 中的实现。

`distance` 方法用于计算两个以数组表示的坐标之间的 Euclidian 距离，此方法被用在判断向量与中心距离，以及判断更新后的中心与更新前的中心的距离中：

```
def distance(point: Array[Double], center: Array[Double]): Double = {
    val dist = point.zip(center).map(pair => (pair._1 - pair._2) * (pair._1
- pair._2)).reduce(_ + _)
    return dist
}
```

`closestCenter` 方法可用于计算与给定点最近的聚类中心，并返回相应聚类中心的序号：

```
def closestCenter(point: Array[Double], centers: Array[Array[Double]]): Int
= {
    var realPoint = point.drop(1)
    var tmp = for(i <- 0 until centers.length) yield (centers(i), i)
    var bestIndex = tmp.map(item => (distance(realPoint, item._1),
item._2)).minBy(item => item._1)._2
    return bestIndex
}
```

分布式考虑

与 Hadoop MapReduce 极为标志性的 `map` 与 `reduce` 方法不同，在 Spark 中完成分布式部分并不需要符合接口并将相应的代码写在固定的方法中，只需要将数据载入为 RDD，并调用符合 RDD API 的属性即可实现分布式，因此，我们可以以一种更为灵活的方式来组织代码。典型的 RDD API 包括 `map`、`keyBy`、`filter`、`reduce` 等，在此处可查阅完整 API 文档。

通过创建 `SparkConf` 实例，我们可以设置包括应用名称，master 节点名称等在内的工程基本信息，并通过此信息初始化 `SparkContext` 来在 Spark 框架下完成相应算法。

```
val conf = new SparkConf().setAppName("Kmeans")
val sc = new SparkContext(conf)
```

我们使用 `sparkContext.textFile` 属性从 HDFS 中夹在文件并作为 RDD。值得注意的是，由于 Spark 所定义的 DAG 计算模型特性，此代码并不直接将文件加载为 RDD，而是记录文件来源，操作时再懒加载相应文件。使用 `cache` 方法可以使相应的文件一旦被加载，便会被缓存到内存中，从而减少数据存取的时间。而由于通过本地随机算法初始化的聚类中心较小（通常只有数个向量），我们使用 `collect` 方法将 RDD 转化为正常的数组类型，提高访问的速度。

```
val points = sc.textFile(pointsFilePath)
        .map(line => (line.split(" ").map(word => word.toDouble))).cache()
var centers = sc.textFile(centersFile)
    .map(line => (line.split(" ").map(word => word.toDouble))).collect
```

通过调用 RDD API 中的 `map` 及 `reduceByKey` 方法，我们将计算每个向量归属的聚类中心以及计算新的聚类中心的工作分散在多个节点上完成，以此实现分布式计算的目标。

```
val temp = points.map(point => (closestCenter(point, centers),
(point.drop(1), 1) ))
val newCenters = temp
    .reduceByKey{case((point1, num1), (point2, num2)) => (addArray(point1,
point2), num1 + num2)}
    .map{case(id, (pointSum, numSum)) => (id, pointSum.map(_ /
numSum))}.collect
var diff : Double = 0
for((id, newCenter) <- newCenters) {
    diff += distance(centers(id), newCenter)
    centers(id) = newCenter.map(item => item)
}
```

依靠以上设计，我们以 Hadoop MapReduce 框架下 $\frac{1}{4}$ 的 代码量完成了适用于 Spark 框架的 KMeans 算法。

运行及结果分析

在测试部分，我们使用实验二中从邮件数据集提取的词向量以及随机生成的聚类中心向量进行测试。根据我们之前的数据清洗方法以及向量换算方法，每个词向量为 100 维，在文件中以换行符分隔，第一位附加了相应向量的序号。每个聚类中心为 100 维，使用随机算法生成，在不声明的前提下，我们默认使用 10 个聚类中心。关于相应向量的具体生成方式以及初始聚类中心的随机选取方式可参考之前的实验报告内容。

sbt 工程构建

为了获得可执行的 .jar 文件，我们需要对完成的 Scala 工程进行打包，sbt 能够帮助我们完成这件事。在实践过程中，我们发现 sbt 对于工程目录的要求较为严格，如果目录中仅仅放一个 .scala 文件是无法完成 `sbt package` 命令的，一个有效的工程目录应该具有以下结构：

```
+-- src/
|   +-- main/
|   |   +-- resources/
|   |   +-- scala/
|   |   |   +-- Kmeans.scala
|   |   +-- java/
+-- build.sbt
```

其中，build.sbt 文件用于描述工程的基本信息，包括名称，使用的配置版本，以及依赖的相关库的版本等，本实验中完成的 build.sbt 文件内容如下：

```
ThisBuild / scalaVersion := "2.11.6"
ThisBuild / organization := "Yeeef CC Emma"
libraryDependencies += "org.apache.spark" %% "spark-core" % "2.4.0"
lazy val hello = (project in file("."))
  .settings(
        name := "Kmeans"
  )
```

完成相应目录的准备之后即可在目录中通过 `sbt package` 命令将工程打包为 .jar 文件，.jar 文件可以在 `./target/scala-2.11` 中找到。

运行 Spark 项目

通过 `spark-submit` 命令可以将代码提交给 Spark 执行，此命令具有众多参数，可在官方文档中查看参数的具体释义。我们所使用的 `master` 参数表示选取 url 为 spark://host0:7077 的节点作为集群的 master 节点。通过以下命令，我们在 Spark 上开始运行之前完成的 KMeans 算法：

```
spark-submit --class "Kmeans" --master spark://host0:7077 ./kmeans_2.11-
0.1.0-SNAPSHOT.jar 6 /user/datapath/doc_vec /user/centerpath/center
/user/tmppath/ 10 0.01
```

我们获取到如下输出内容，其中包含了我们计算的运行时间输出：

```
2018-11-27 18:46:35 INFO  SparkContext:54 - Running Spark version 2.4.0
2018-11-27 18:46:35 INFO  SparkContext:54 - Submitted application: Kmeans
2018-11-27 18:46:35 INFO  SecurityManager:54 - Changing view acls to: root
2018-11-27 18:46:35 INFO  SecurityManager:54 - Changing modify acls to:
root
2018-11-27 18:46:35 INFO  SecurityManager:54 - Changing view acls groups
to:
2018-11-27 18:46:35 INFO  SecurityManager:54 - Changing modify acls groups
to:
```

```
2018-11-27 18:46:35 INFO  SecurityManager:54 - SecurityManager:
authentication disabled; ui acls disabled; users  with view permissions:
Set(root); groups with view permissions: Set(); users  with modify
permissions: Set(root); groups with modify permissions: Set()
# Details omitted
================================
Iter 17 times

Time elapsed: 67.735s

================================
# Details omitted
```

经过检验，此算法的输出与之前 Hadoop MapReduce 框架下的 KMeans 算法输出一致。

与 Hadoop MapReduce 的对比

使用相同思路的算法以及相同的数据输入，我们能够对比 Spark 与 Hadoop MapReduce 的性能。在本部分，我们将通过改变聚类中心数量获取多组数据，来对比 Spark 框架与 Hadoop MapReduce 框架的性能差异。

我们通过在放置初始聚类中心的文件头部删去 0 / 2 / 4 / 6 行的方法，来获取 k 值选取 10 / 8 / 6 / 4 时的初始聚类中心，并通过如下命令运行之前实验中完成的 KMeans 算法，从而进行对比。

```
hadoop jar KMeans.jar Kmeans /user/datapath/doc_vec /user/centerpath/center
/user/tmppath 0.01 30
```

我们记录了不同的 k 值选取下的迭代轮数以及运算时间，并记录在如下的表格中。

k 值	迭代轮数	Hadoop 运行时间（秒）	Spark 运行时间（秒）
4	17	737.258	44.54
6	12	594.351	44.732
8	20	1098.081	66.069
10	17	977.733	67.735

可以看到，在相同的输入及算法条件下，Spark 的运行速度远快于 Hadoop MapReduce 的速度，使用如下的速度对比图表能够更清晰地展示两者速度的关系。

我们观察到，平均意义上 Spark 比 Hadoop MapReduce 的性能提升达到了 15 倍左右，这可以说是 Spark 决定性的性能优势。由于省去了 Hadoop MapReduce 作业之间文件 IO 的时间浪费，以及 KMeans 算法中对 map reduce 过程的循环调用，使得 Spark 相对于 Hadoop MapReduce 在 KMeans 算法上有令人惊异的性能表现。

相关问题

- 运行 `spark-shell` 后，出现警告

  ```
  WARN  NativeCodeLoader:62 - Unable to load native-hadoop library for
  your platform... using builtin-java classes where applicable
  ```

  - 在 `$SPARK_HOME/conf/spark-env.sh` 中添加

    ```
    export LD_LIBRARY_PATH=$HADOOP_HOME/lib/native/
    ```

- 在 spark 中运行的程序可以读取本地文件，也可以读取 hdfs 中的文件
  - 默认读取 hdfs 中的文件，我们也可以添加前缀 `hdfs://` 来特别指明
  - 若要读取本地文件
    - 本地文件必须在所有 work / master 中都存在在同一路径
    - 添加前缀 `file://` 来特别指明
- 一开始在进行 WordCount 简单实践中，会出现 `unexpected exception` 的错误
  - 在 `build.sbt` 中没有指定正确的 `scalaVersion`, 修改后错误解决