Санкт-Петербургский Государственный Университет
Факультет Прикладной математики — процессов управления
Кафедра технологий программирования и информационных систем

Санникова Елена Сергеевна

Курсовая работа

Создание поисковой машины с помощью фреймворка Spark

Научный руководитель,
кандидат физ.-мат. наук,
доцент
Добрынин В. Ю.

Санкт-Петербург 2016

Содержание

1. Введение.....3
2. Введение в анализ данных с помощью Apache Spark.....4
2.1. Обобщенный обзор Apache Spark.....4
2.2. Основные компоненты Spark.....4
2.3. Загрузка и настройка.....6
2.4. Пример запуска программы в командной оболочке Spark для Python.....7
2.5. Основные понятия Spark.....8
3. Создание поисковой машины.....10
3.1. Исследование и обработка коллекции OHSUMED.....10
3.2. Построение индекса коллекции документов.....12
3.3. Обработка поискового запроса.....13
4. Итоги.....14
5. Список литературы.....15

1. Введение

В современном мире область анализа данных развивается очень стремительно. Только задумайтесь, сколько всего мы загружаем на серверы: видео, фотографии, текстовые сообщения, отправленные друзьям, оставляем комментарии на разных сайтах и т. д. Компьютерам приходится хранить все больше и больше данных. Естественно, такие крупные компании, как Google, Yahoo, Amazon и Microsoft одни из первых столкнулись с проблемой экспоненциального роста данных. Им приходилось обрабатывать терабайты и петабайты данных. Имеющиеся на тот момент инструменты оказались непригодны для обработки такого объема данных.

Первым удачным решением проблемы стал фреймворк MapReduce, предложенный и впоследствии активно внедряемый компанией Google. В MapReduce для вычисления некоторого набора распределенных задач используется большое количество компьютеров («нодов»), объединенных в кластер. Благодаря распределенной обработке и свертке данных фреймворк позволил довольно просто обрабатывать большие объемы данных большим количеством серверов.

Фреймворк вызвал большой интерес других компаний, для которых эта проблема оставалась актуальной. Дуг Каттинг разглядел открывающиеся возможности и принялся за разработку версии MapReduce с открытым исходным кодом, которую назвал Hadoop[1]. Вскоре это начинание поддержали другие крупные компании, как Yahoo. Сегодня Hadoop составляет основную часть вычислительной инфраструктуры многих работающих в веб компаний.

Вслед за Hadoop фреймворком следующего поколения становится Apache Spark[2]. Появившись в 2010 году, за очень короткое время он получил повсеместное распространение. Spark оказался намного быстрее Hadoop, проще в использовании и включил в себя такие необходимые инструменты, которые позволили создавать интерактивные приложения, приложения потоковой обработки данных, машинного обучения и обработки графов.

В данной работе мы познакомимся с фреймворком Spark, индексируем с его помощью коллекцию документов и построим простой векторный поиск документов по запросу с ранжированной выдачей.

1 http://hadoop.apache.org/

2 http://spark.apache.org/

2. Введение в анализ данных с помощью Apache Spark

2.1. Обобщенный обзор Apache Spark

Apache Spark - это универсальная и высокопроизводительная кластерная вычислительная платформа.

Относительно скорости, Spark превосходит популярную модель MapReduce. При обработке больших массивов данных скорость очень важна, т. к. именно она позволяет работать в интерактивном режиме. Кроме того, преимуществом Spark является способность выполнять вычисления в памяти. Целью фреймворка была возможность объединить в себе такие инструменты, как интерактивные запросы, циклические алгоритмы, потоковую обработку, создание распределенных систем.

Spark стал очень доступным инструментом, предлагая простой API на Java, Scala, Python и SQL, а так же богатую коллекцию встроенных библиотек.

2.2. Основные компоненты Spark

Проект Spark включает множество тесно связанных компонентов. Рассмотрим основные из них.

Spark Core реализует основные функциональные возможности фреймворка Spark, включая компоненты, осуществляющие планирование заданий, управление памятью, обработку ошибок, взаимодействие с системами хранения данных и многие другие. Spark Core является также основой API устойчивых распределенных наборов данных (Resilient Distributed Datasets, RDD) - базовой абстракции Spark. Наборы данных RDD представляют собой коллекции элементов, распределенных между множеством вычислительных узлов, которые могут обрабатываться параллельно. Spark Core предоставляет множество функций управления такими коллекциями.

Spark SQL - пакет для работы со структурированными данными. Позволяет извлекать данные с помощью инструкций на языке SQL и его диалекте Hive Query Language (HQL). Поддерживает множество источников данных, включая таблицы Hive, Parquet и JSON. В дополнение к интерфейсу SQL компонент Spark SQL позволяет смешивать в одном приложении запросы

SQL с программными конструкциями на Python, Java и Scala, поддерживаемыми абстракцией RDD, и таким способом комбинировать SQL со сложной аналитикой. Подобная тесная интеграция с богатыми возможностями вычислительной среды выгодно отличает Spark SQL от любых других инструментов управления данными. Spark SQL был добавлен в стек Spark в версии 1.0. Первой реализацией поддержки SQL в Spark стал проект Shark, созданный в Калифорнийском университете, Беркли. Этот проект представлял собой модификацию Apache Hive, способную выполняться под управлением Spark. Позднее ему на смену пришел компонент Spark SQL, имеющий более тесную интеграцию с механизмом Spark и API для разных языков программирования.

Spark Streaming - компонент Spark для обработки потоковых данных. Примерами источников таких данных могут служить файлы журналов, заполняемые действующими веб-серверами, или очереди сообщений, посылаемых пользователями веб-служб. Spark Streaming имеет API для управления потоками данных, который близко соответствует модели RDD, поддерживаемой компонентом Spark Core, что облегчает изучение самого проекта и разных приложений обработки данных, хранящихся в памяти, на диске или поступающих в режиме реального времени. Прикладной интерфейс (API) компонента Spark Streaming разрабатывался с прицелом обеспечить такую же надежность, пропускную способность и масштабируемость, что и Spark Core.

В состав Spark входит библиотека **MLlib**, реализующая механизм машинного обучения (Machine Learning, ML). MLlib поддерживает множество алгоритмов машинного обучения, включая алгоритмы классификации (classification), регрессии (regression), кластеризации (clustering) и совместной фильтрации (collaborative filtering), а также функции тестирования моделей и импортирования данных. Она также предоставляет некоторые низкоуровневые примитивы ML, включая универсальную реализацию алгоритма оптимизации методом градиентного спуска. Все эти методы способны работать в масштабе кластера.

GraphX - библиотека для обработки графов (примером графа может служить граф друзей в социальных сетях) и выполнения параллельных вычислений. Подобно компонентам Spark Streaming и Spark SQL, GraphX дополняет Spark RDD API возможностью создания ориентированных графов с

произвольными свойствами, присваиваемыми каждой вершине или ребру. Также GraphX поддерживает разнообразные операции управления графами (такие как subgraph и rnapVertices) и библиотеку обобщенных алгоритмов работы с графами (таких как алгоритмы ссылочного ранжирования PageRank и подсчета треугольников).

Внутренняя реализация Spark обеспечивает эффективное масштабирование от одного до многих тысяч вычислительных узлов. Для достижения такой гибкости Spark поддерживает большое многообразие **диспетчеров кластеров** (cluster managers), включая Hadoop YARN, Apache Mesos, а также простой диспетчер кластера, входящий в состав Spark, который называется Standalone Scheduler. При установке Spark на чистое множество машин на начальном этапе с успехом можно использовать Standalone Scheduler. При установке Spark на уже имеющийся кластер Hadoop YARN или Mesos можно пользоваться встроенными диспетчерами этих кластеров.

2.3. Загрузка и настройка

Рассмотрим процесс установки и запуска Spark в локальном режиме на 1 компьютере. Откроем страницу загрузки http://spark.apache.org/downloads.html, выберем тип пакета «Pre-built for Hadoop 2.6 and later» и тип загрузки «Direct Download». Затем скачаем файл с именем «spark-1.6.1-bin-hadoop2.6.tgz» по ссылке ниже. Распакуем архивированный файл с фреймворком Spark. Перейдем в каталог и посмотрим его содержание:

- README.md — содержит инструкции по настройке;
- bin — содержит выполняемые файлы;
- core, streaming, python … — содержат исходный код основных компонентов проекта Spark;
- examples - содержит примеры реализации некоторых распространенных задач.

Попробуем воспользоваться командными оболочками Spark для Python. Запустим пример из examples, а затем начнем реализовывать свою программу.

2.4. Пример запуска программы в командной оболочке Spark для Python

Рассмотрим, как запускать программу на Python в Spark на простой программе «wordcount.py», которая подсчитывает частоту встречаемости слов в тексте.

Программа «wordcount.py»:

```
def run(sc):
 rdd_lines = sc.textFile("README.md")
 rdd_words = rdd_lines.flatMap(lambda line: line.split())
 rdd_mapper_output = rdd_words.map(lambda w : (w, 1))
 rdd_reducer_output = rdd_mapper_output.reduceByKey(lambda a, b: a+b)
 rdd_reducer_output.saveAsTextFile("wordcount_result")
```

Сначала запустим одну из командных оболочек Spark. Чтобы запустить командную оболочку для Python, которую также называют PySpark Shell, перейдем в каталог установки Spark и выполним команду:

bin/pyspark

Запустим питоновскую консоль, передав аргумент «--py-files»:

bin/pyspark --py-files wordcount.py

После этого «wordcount» будет находиться в «PYTHONPATH» и его можно будет импортировать.

Когда оболочка запустится и появится приглашение к вводу (>>>), введем:

from wordcount import run

run(sc)

После выполнения программы в каталоге появится каталог «wordcount_result», в котором будут лежать несколько файлов с названиями «part-0000*» и файл «_SUCCESS», говорящий об успешном завершении записи результатов. В файлах «part-0000*» записаны пары: (слово, сколько раз оно встречается в README.md).

2.5. Основные понятия Spark

В данном параграфе рассмотрим основные понятия и приемы программирования, а заодно разберем пример программы «wordcount.py».

В Spark вычисления выражаются в виде операций с распределенными коллекциями, которые автоматически расспараллеливаются на ноды. Эти коллекции называются Resilient Distributed Datasets (RDD). В «wordcount.py» определяется переменная rdd_lines, представляющая набор RDD, созданный из текстового файла. С набором RDD можно выполнять разнообразные параллельные операции.

Любое приложение на основе Spark состоит из программы-драйвера (driver program), который запускает различные параллельные операции в кластере. Драйвер содержит функцию main приложения и определяет распределенные наборы данных, а затем применяет к ним различные операции. В нашем примере роль драйвера выполняет сама командная оболочка Spark, благодаря чему можно просто вводить желаемые операции.

Драйвер обращается к Spark посредством объекта SparkContext, представляющего соединение с вычислительным кластером. Командная оболочка Spark автоматически создает объект SparkContext в виде переменной с именем sc. Имея объект SparkContext, можно создавать наборы RDD. В примере с этой целью вызывался метод **sc.textFile()**, создающий RDD, который представляет строки из текстового файла.

Поддерживаются два способа создания наборов RDD: путем загрузки внешних наборов данных и распределением коллекций в программе-драйвере. Самый простой способ - взять существующую коллекцию и передать ее методу **parallelize()** объекта SparkContext. Такой подход удобно использовать при изучении Spark, поскольку позволяет быстро создать собственный набор RDD в командной оболочке и приступить к выполнению операций с ним. После создания набора можно приступать к выполнению разнообразных операций.

Бывают два вида операций: преобразования (transformations) и действия (actions). Преобразования создают новые наборы RDD на основе существующих. Примером служит операция фильтрации данных по заданному условию:

```
spark_lines = rdd_lines.filter(lambda line: "Spark" in line)
```

spark_lines будет представлять собой RDD из строк, в которых есть слово «Spark».

Действия, напротив, вычисляют результат, не создавая новых наборов RDD, и возвращают его программе-драйверу или сохраняют во внешнем хранилище. Примером действия может служить вызов метода **first()**, который возвращает первый элемент RDD:

```
spark_lines.first()
```

Преобразования и действия отличаются способом обработки наборов RDD. Даже при том, что новый набор RDD можно создать в любой момент, Spark откладывает фактическое его создание до момента первого обращения к нему.

Рассмотрим часто используемые преобразования и действия. Преобразование **map()** принимает функцию и применяет ее к каждому элементу в наборе RDD, а результат этой функции сохраняется как значение элемента в новом наборе RDD. Иногда бывает желательно произвести несколько новых элементов для каждого исходного элемента. Сделать это можно с помощью операции **flatMap()**. Подобно преобразованию **map()**, преобразование **flatMap()** принимает функцию и вызывает ее для каждого элемента в исходном наборе RDD. Однако вместо набора RDD итераторов **flatMap()** возвращает набор RDD с элементами, получаемыми с применением всех итераторов. Так в нашем примере **flatMap()** разбивает исходные строки на слова, а **map()** каждому слову ставит в соответствие пару: (слово, 1).

Преобразование **reduceByKey()** принимает функцию и использует ее для объединения значений. **reduceByKey()** запускает несколько параллельных операций свертки (reduce), по одной для каждого ключа в наборе, где каждая операция объединяет значения с одинаковыми ключами. В нашем примере **reduceByKey()** вернет результирующий RDD, сложив все единички для каждого слова.

Сохранить содержимое набора RDD можно вызовом действий **saveAsTextFile()**, **saveAsSequenceFile()** и ряда других, сохраняющих данные в разных поддерживаемых форматах.

Больше методов и операций над RDD можно найти в документации Spark[3].

3 http://spark.apache.org/docs/latest/programming-guide.html

3. Создание поисковой машины

Теперь, используя все то, что мы уже знаем, напишем программу, которая в среде Spark проиндексирует коллекцию документов. Затем, используя построенный индекс, хотим возвращать список документов, удовлетворяющих запросу, в порядку ранжирования. Но для начала познакомимся с самой коллекцией.

Примечание: Коллекция и весь код, который использовался и описан в данной работе находится в публичном репозитории на GitHub[4].

3.1. Исследование и обработка коллекции OHSUMED

Коллекция OHSUMED представляет собой набор из 348,566 статей из медицинской информационной он-лайн базы данных MEDLINE. Статьи являются выписками из 270 медицинских журналов за пятилетний период (1987-1991 годы).

Пример статьи из коллекции:

```
.I 1
.U
87049087
.S
Am J Emerg Med 8703; 4(6):491-5
.M
Allied Health Personnel/*; Electric Countershock/*; Emergencies; Emergency Medical
Technicians/*; Human; Prognosis; Recurrence; Support, U.S. Gov't, P.H.S.; Time Factors;
Transportation of Patients; Ventricular Fibrillation/*TH.
.T
Refibrillation managed by EMT-Ds: incidence and outcome without paramedic back-up.
.P
JOURNAL ARTICLE.
.W
Some patients converted from ventricular fibrillation to organized rhythms by
defibrillation-trained ambulance technicians (EMT-Ds) will refibrillate before hospital
arrival. The authors analyzed 271 cases of ventricular fibrillation managed by EMT-Ds
working without paramedic back-up. Of 111 patients initially converted to organized
rhythms, 19 (17%) refibrillated, 11 (58%) of whom were reconverted to perfusing rhythms,
including nine of 11 (82%) who had spontaneous pulses prior to refibrillation. Among
patients initially converted to organized rhythms, hospital admission rates were lower
for patients who refibrillated than for patients who did not (53% versus 76%, P = NS),
although discharge rates were virtually identical (37% and 35%, respectively). Scene-to-
hospital transport times were not predictively associated with either the frequency of
refibrillation or patient outcome. Defibrillation-trained EMTs can effectively manage
refibrillation with additional shocks and are not at a significant disadvantage when
paramedic back-up is not available.
.A
Stults KR; Brown DD.
```

Поля в статье обозначают следующее: «.I» - последовательный

4 https://github.com/el-san59/Course_Work_Spark.git

идентификатор; «.U» - MEDLINE-идентификатор; «.M» - термины MeSH; «.T» - Название статьи; «.P» - Тип публикации; «.W» - Аннотация; «.A» - Авторы; «.S» - Источник.

Чтобы упростить обработку коллекции, перепишем ее в формат JSON. И разобъем коллекцию на JSON-файлы, содержащие по 300 статей. Получим 1163 JSON-файла, размера в среднем по 400 Кб. Так будет выглядеть статья после обработки:


```
{ ".I": "1",
   ".M": "Allied Health Personnel/*; Electric Countershock/*; Emergencies; Emergency Medical Technicians/*; Human; Prognosis; Recurrence; Support, U.S. Gov't, P.H.S.; Time Factors; Transportation of Patients; Ventricular Fibrillation/*TH.\r",
  ".A": [ "Stults", "Brown" ],
  ".P": "JOURNAL ARTICLE.\r",
  ".S": "Am J Emerg Med 8703; 4(6):491-5\r",
  ".T": "Refibrillation managed by EMT-Ds: incidence and outcome without paramedic back-up.\r",
  ".U": "87049087\r",
   ".W": "Some patients converted from ventricular fibrillation to organized rhythms by defibrillation-trained ambulance technicians (EMT-Ds) will refibrillate before hospital arrival. The authors analyzed 271 cases of ventricular fibrillation managed by EMT-Ds working without paramedic back-up. Of 111 patients initially converted to organized rhythms, 19 (17%) refibrillated, 11 (58%) of whom were reconverted to perfusing rhythms, including nine of 11 (82%) who had spontaneous pulses prior to refibrillation. Among patients initially converted to organized rhythms, hospital admission rates were lower for patients who refibrillated than for patients who did not (53% versus 76%, P = NS), although discharge rates were virtually identical (37% and 35%, respectively). Scene-to-hospital transport times were not predictively associated with either the frequency of refibrillation or patient outcome. Defibrillation-trained EMTs can effectively manage refibrillation with additional shocks and are not at a significant disadvantage when paramedic back-up is not available.\r" }
```


Поставим перед собой задачу: хотим выдавать номера наиболее подходящих документов на запросы пользователя следующего вида:

--query "Текст запроса" --weights 0.1,0.5,0.4

где --weights — разделённые запятой веса (сумма весов равна единице). Первым идет вес авторов, вторым вес названия, третьим вес аннотации. Т. е. если пользователь хочет, чтобы текст запроса, который он ввел первым параметром, преимущественно находился в заголовке, то параметр весов он укажет примерно такой: 0.1,0.8,0.1.

3.2. Построение индекса коллекции документов

Будем строить индекс используя только поля, содержащие авторов, названия и аннотации статей. Пусть files — это список наших JSON-файлов. Создадим набор RDD под названием data:

```
data = sc.parallelize(files)
```

Функции flatMap() передадим функцию mapper(), которая принимает JSON-файл и возвращает нам пары: (key, value), где key — это слово, которое мы встретили, а value — это пара, первый элемент которой — идентификатор статьи, в которой встретили слово, а второй — метка, показывающая где мы это слово встретили (в названии, в аннотации или в списке авторов):

```
data = data.flatMap(lambda shard: mapper(shard))
```

Далее сгруппируем полученные пары по ключу, т. е. каждому слову сопоставим список пар из идентификаторов статей, где это слово встречалось, и из метки, в какой части статьи оно было обнаружено:

```
data = data.groupByKey()
```

После этого преобразования каждая пара обладает уникальным ключом. Передадим полученный RDD функции map(), которая будет принимать функцию reducer():

```
data = data.map(lambda word: reducer(word))
```

Функция reducer() принимает наши пары (key, value), где value — это уже список пар, а возвращает пары (key, value), где key остается тем же самым словом, а вот value меняется на тройки, первая и вторая часть все так же идентификатор статьи и метка, а третья часть — мера tf-idf слова в документе. Пусть D — множество документов в коллекции, $f_{t,d}$ — частота слова t в документе d. Мера tf-idf слова высчитывается по формулам:

$$tfidf(t,d,D) = tf(t,d) * idf(t,D)$$

где *tf* — term frequency:

$$tf(t,d) = 1 + \log(f_{t,d})$$

а *idf* — inverse document frequency:

$$idf(t,D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

Теперь мы имеем инвертированный индекс всей коллекции.

Отсортируем его по ключу (по алфавиту):

```
data = data.sortByKey()
```

И запишем результаты в коллекцию файлов:

```
data.foreach(print_res)
```

Например, для слова «warranty» у нас будет записано следующее:

```
(u'warranty', [(u'33929', 'w', 11.375288513202417), (u'160202', 'w',
11.375288513202417), (u'307715', 'w', 11.375288513202417), (u'89588', 't',
11.375288513202417)])
```

На этом работа Spark завершается. Далее мы обойдемся без него. Начнем обрабатывать пользовательские запросы.

3.3. Обработка поискового запроса

Напомним что запрос выглядит следующим образом:

```
--query "Текст запроса" --weights 0.1,0.5,0.4
```

Находим по очереди все слова из запроса. Для каждого слова обрабатываем его индекс (список троек). Каждый элемент списка порождает пару (key, value), где key — идентификатор статьи, а value — мера *tf-idf*, умноженная на вес соответствующей метки из рассматриваемой тройки. Далее группируем полученные пары по ключу (по идентификатору статьи) и для каждого суммируем значения. Теперь, отсортировав по значению, мы получим ранжированную выдачу документов а запрос.

Для запроса «--query "diastolic pneumatic compression boot" --weights 0.1,0.5,0.4» программа выдала в первой тройке статьи с такими заголовками:

- "Successful treatment of osteomyelitis and soft tissue infections in ischemic diabetic legs by local antibiotic injections and the end-**diastolic pneumatic compression boot.**\r";
- "**Pneumatic** sequential-**compression boots** compared with aspirin prophylaxis of deep-vein thrombosis after total knee arthroplasty.\r";
- "Acute compartment syndrome caused by a malfunctioning **pneumatic-compression boot.** A case report.\r".

4. Итоги

В ходе практической работы мы освоили основы работы в Spark. Убедились, что это весьма удобный и мощный инструмент, использующий единую парадигму программирования вместо смеси инструментов. Очень быстрый, простой в использовании, постоянно обновляющийся и расширяющийся в инструментарии Spark может стать прекрасным помощником в современном анализе больших объемов данных.

Список литературы:

1. Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia «Learning Spark»
2. Chuck Lam «Hadoop in action»
3. Christopher D. Manning, Prabhakar Raghavan, Hinrich Schutze «Introduction to Information Retrieval»
4. Spark Programming Guide http://spark.apache.org/docs/latest/programming-guide.html