Введение

Предлагается решить задачу 1 используя 2 различных Spark API:

1. RDD API,
2. DataFrame API.

а также задачу 2, используя любое из 2 API.

Цель задания, убедиться насколько DataFrame API быстрее и удобнее чем RDD.

- ❏ Репозиторий для сдачи: http://gitlab.atp-fivt.org/hobod2020/…
- ❏ Ветки:
  - ❏ hobsparktask1 - для кода задачи 1 на RDD,
  - ❏ hobsparktask2 - для кода задачи 1 на DF,
  - ❏ hobsparktask3 - для кода задачи 2 (на DF).

Сроки

- ➢ Мягкий deadline: 04.04, 23:59.
- ➢ Жесткий deadline: 11.05, 23:59.

Задача 1

Исходные данные

Данные лежат в HDFS.

- Полный датасет: `/data/twitter/twitter_sample.txt` ***(при коммите в систему указывайте в коде этот датасет)***
- Частичная выборка: `/data/twitter/twitter_sample_small.txt`

Формат данных:

```
user_id \t follower_id
```

Условие задачи

Дан ориентированный граф. Необходимо найти длину кратчайшего пути между вершинами 12 и 34 графа, реализовав алгоритм "Поиск в ширину". Если кратчайших путей несколько, выведите первый.

Данную задачу нужно:

решить двумя способами - с помощью RDD и с помощью DF API

замерить CPU time (не wall time поскольку он измеряет время с учётом загруженности кластера).

Для сдачи задачи нужно не только пройти тесты, но и прислать (закоммитить в репозиторий) 2 числа - замеры времени работы каждого способа.

Обратите внимание на критерий остановки алгоритма. В рамках оптимизации вы можете остановить программу раньше, чем закончится поиск в ширину т.к. нам достаточно одно пути.

Выходной формат: последовательность вершин (учитывая начало и конец), разделенных запятой, без пробелов. Например, путь «12 -> 42 -> 34» должен быть напечатан как: `12,42,34`.

Пример вывода

```
12,42,57,34
```

По возможности, необходимо избегать написания UDF, поскольку UDF ухудшают производительность. Вместо этого внимательно изучите возможности pyspark.sql.functions. Вам точно пригодится этот модуль.

Стартовый фрагмент кода

От этого фрагмента кода можно отталкиваться при решении задачи. Этот код не эффективный поэтому он не будет работать в системе проверки. Его цель - дать понимание, от чего отталкиваться в задаче.

```
def parse_edge(s):
  user, follower = s.split("\t")
  return (int(user), int(follower))

def step(item):
  prev_v, prev_d, next_v = item[0], item[1][0], item[1][1]
  return (next_v, prev_d + 1)

def complete(item):
  v, old_d, new_d = item[0], item[1][0], item[1][1]
  return (v, old_d if old_d is not None else new_d)

n = 400  # number of partitions
edges = sc.textFile("/data/twitter/twitter_sample_small.txt").map(parse_edge)
forward_edges = edges.map(lambda e: (e[1], e[0])).partitionBy(n).persist()

x = 12
d = 0
distances = sc.parallelize([(x, d)]).partitionBy(n)
while True:
  candidates = distances.join(forward_edges, n).map(step)
  new_distances = distances.fullOuterJoin(candidates, n).map(complete, True).persist()
  count = new_distances.filter(lambda i: i[1] == d + 1).count()
  if count > 0:
    d += 1
    distances = new_distances
  else:
    break
```

Код для создания SparkContext.

```
from pyspark import SparkContext, SparkConf

config = SparkConf().setAppName("my_super_app").setMaster("local[3]")  # конфиг, в котором указываем название приложения и режим выполнения (local[*] для локального запуска, yarn для запуска через YARN). В систему сдаём код с мастером YARN.
sc = SparkContext(conf=config)  # создаём контекст, пользуясь конфигом
```

Задача 2

Исходные данные

➢ Статьи Википедии: `/data/wiki/en_articles_part`. Данные лежат в HDFS.

Формат данных:

```
article ID <tab> article text
```

➢ Список стоп-слов: `/data/wiki/stop_words_en-xpo6.txt`.

Формат данных: одно стоп-слово на строчку

```
...
wherein
whereupon
wherever
...
```

Задача состоит в извлечении коллокаций. Это комбинации слов, которые часто встречаются вместе. Например, «High school» или «Roman Empire». Чтобы найти совпадения, нужно использовать метрику NPMI (нормализованная точечная взаимная информация).

PMI двух слов a и b определяется как

$$PMI(a, b) = ln\left(\frac{P(a,b)}{P(a)\cdot P(b)}\right)$$

, где *P(ab)* - вероятность двух слов, идущих подряд, а P (a) и P (b) - вероятности слов a и b соответственно.

Вам нужно будет оценить вероятности встречаемости слов, то есть

P(a)=num_of_occurrences_of_word_"a" / num_of_occurrences_of_all_words
P(ab)=num_of_occurrences_of_pair_"ab"‘ / num_of_occurrences_of_all_pairs

- *total_number_of_words - общее кол-во слов в тексте*
- *total_number_of_word_pairs - общее кол-во пар*

- **"Roman Empire";** предположим, что это уникальная комбинация, и за каждым появлением «Roman» следует «Empire», и, наоборот, каждому появлению «Empire» предшествует «Roman». В этом случае «P (ab) = P (a) = P(b)», поэтому «PMI (a, b) = -lnP(a) = -lnP(b)». Чем реже встречается эта коллокация, тем больше значение PMI.
- **"the doors";** предположим, что «the» может встретится рядом с любым словом. Таким образом, «P (ab) = P (a) * P (b)» и «PMI (a, b) = ln 1 = 0".
- **«green idea / sleeps furiously»;** когда два слова никогда не встречаются вместе, «P (ab) = 0» и «PMI (a, b) = -inf».

NPMI вычисляется как «$NPMI(a, b) = -\frac{PMI(a,b)}{ln\, P(a,b)}$». Это нормализует величину в диапазон [-1; 1].

Условие задачи

Найти самые популярные коллокации в Википедии. Обработка данных:

- ➢ При парсинге отбрасывайте все символы, которые не являются латинскими буквами: `text = re.sub("^\W+|\W+$", "", text)`
- ➢ приводим все слова к нижнему регистру;
- ➢ удаляем все стоп-слова (даже внутри биграммы т.к. “at evening” имеет ту же семантику что и “at the evening”);
- ➢ биграммы объединить символом нижнего подчеркивания "_";
- ➢ работаем только с теми биграммами, которые встретились не реже 500 раз (т.е. проводим все необходимые join'ы и считаем NPMI только для них).
- ➢ общее число слов и биграмм считать до фильтрации.

Для каждой биграммы посчитать NPMI и вывести на экран (в STDOUT) TOP-39 самых популярных коллокаций, отсортированных по убыванию значения NPMI. ***Само значение NPMI выводить не нужно.***

Пример вывода

```
roman_empire
south_africa
```

Пример вывода на sample-datacete (со значениями NPMI)

```
19th_century      0.757464166177
20th_century      0.751460453349
references_external     0.731826941011
soviet_union      0.727806412183
air_force   0.705773204264
baseball_player   0.691711138551
university_press  0.687424532005
roman_catholic    0.683677693663
united_kingdom    0.68336461567
```

Подсказка: если вы все сделаете правильно, «roman_empire» и «south_africa» будут в ответе.

Дополнительные комментарии

1. Данных немного поэтому есть соблазн на каком-нибудь этапе решения сделать `take()` или `collect()`, сконвертировав RDD / DF в обычный Python-объект. Конечно, с точки зрения API, работать с обычными объектами привычнее. Но т.к. обычные объекты из коробки не отвечают требованиям высокодоступности и распределённости, такое решение учитываться не будет.
2. Помните, что по возможности необходимо избегать написания UDF, вместо этого внимательно изучите возможности `pyspark.sql.functions`. Вам точно пригодится этот модуль.
3. Для сдачи задания нужно закоммитить в Git PySpark-код, а в run.sh прописать команду для его запуска (`spark2-submit <my_code.py>`).

4. В отличие от ДЗ по MapReduce, логи Spark перенаправлять в /dev/null ***не нужно***.
5. Для локальной отладки кода можно использовать Docker-контейнер.

Технические комментарии

1. Если Spark не может найти свободный порт для UI и выводит ошибки вида:

```
20/04/26 15:10:18 WARN util.Utils: Service 'SparkUI' could not bind on port 4040. Attempting
port 4041.
20/04/26 15:10:18 WARN util.Utils: Service 'SparkUI' could not bind on port 4041. Attempting
port 4042.
20/04/26 15:10:18 WARN util.Utils: Service 'SparkUI' could not bind on port 4042. Attempting
port 4043.
20/04/26 15:10:18 WARN util.Utils: Service 'SparkUI' could not bind on port 4043. Attempting
port 4044.
```

(всего возможно 16 попыток, после чего приложение падает).

Запустите spark2-submit с указанием своего порта, не лежащего в промежутке [4041, 4056]. Например:

```
spark2-submit --conf spark.ui.port=5555
```