|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования **«МИРЭА − Российский технологический университет»**  **РТУ МИРЭА** |

**Институт информационных технологий (ИИТ)**

**Кафедра прикладной математики (ПМ)**

**ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ**

по дисциплине «Системы управления данными»

**Практическое занятие № 5**

|  |  |  |  |
| --- | --- | --- | --- |
| Студент группы ИНБО-01-17 | *ИМБО-02-22, Ким Кирилл Сергеевич* | (подпись) | |
| Преподаватель | *Алексеева Екатерина Сергеевна, преподаватель* | (подпись) | |
| Отчет представлен | «\_\_\_»\_\_\_\_\_\_\_\_202\_\_г. | |  | |

Москва 2025 г.

1. Практическая работа №5
   1. Цель работы

В этой лабораторной мы будем использовать оболочку PySpark для изучения процесса работы в PySpark. Сначала нам стоит получить практический опыт работы с исходной оболочкой. В следующей лабораторной работе мы настроим оболочку для работы с Jupyter и рассмотрим, как работать с PySpark на Jupyter.

## Настройка среды PySpark и запуск оболочки

Настройте переменные среды, чтобы открыть оболочку Pyspark в ее собственных настройках

### Измените настройки запуска оболочки bash

#### Откройте файл .bashrc для редактирования из вашего домашнего каталога. Вы можете использовать любой инструмент редактирования по вашему выбору. Здесь мы покажем, как использовать KWrite.

#### Откройте KWrite, нажав на значок KWrite text edition на панели инструментов в правом нижнем углу экрана

###### 

#### Перейдите в свой домашний каталог

###### 

#### Нажмите на значок гаечного ключа, чтобы изменить настройки

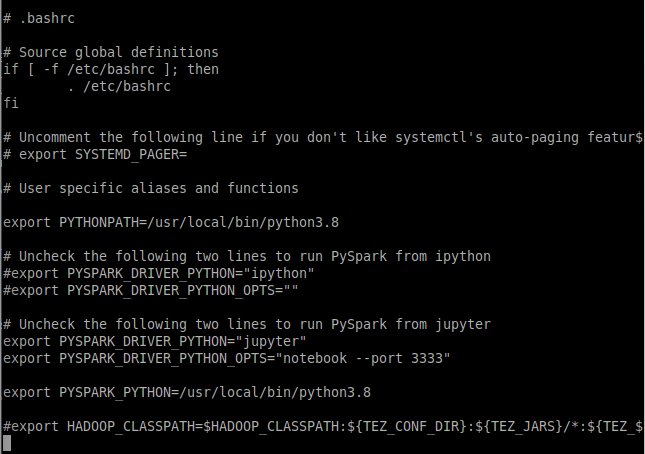
###### 

#### Установите флажок «Показывать скрытые файлы»

###### 

#### Выберите файл .bashrc и отредактируйте файл так, чтобы включить режим ipython (обратите внимание на измененные версии python)

###### 



#### Символ # в начале строки указывает на то, что строка является комментарием. Удалите # комментарий из двух строк настройки для режима python. Когда # будет удален, KWrite изменит цвет текста, как показано выше, чтобы указать, что команды будут выполнены. Убедитесь, что включены только настройки ipython. Настройки Jupyter должны оставаться закомментированы с помощью # в начале строки

#### Обычно файл .bashrc выполняется всякий раз, когда вы запускаете новый сеанс терминала, и настройки связаны только с этим терминалом. Чтобы применить измененные настройки, закройте текущий терминал и откройте новый терминал. В качестве альтернативы, вы также можете выполнить следующую команду из текущего терминала для выполнения файла .bashrc

[student@localhost ~]$ source .bashrc

### Запустите оболочку pyspark

[student@localhost ~]$ pyspark

### 

### Команда pyspark на самом деле представляет собой скрипт, который запускает различные настройки, включая создание SparkContext и SparkSession. Ваш экран должен быть похож на приведенную ниже иллюстрацию:

###### 

### Убедитесь, что SparkContext и SparkSession доступны

#### Запустите sc и spark из оболочки

In [ ]: sc

In [ ]: spark

###### 

### Выйдите из оболочки с помощью команды exit() или Ctrl-d

In [3]: exit()

## Просмотрите текстовый файл в HDFS с помощью Hue и HDFS CLI

### Найдите и скопируйте alice\_in\_wonderland.txt файл в ваш домашний каталог HDFS

#### Перейдите к директории /home/student/Data

[student@localhost ~]$ cd /home/student/Data

#### Откройте alice\_in\_wonderland.txt любым редактором на ваш выбор. KWrite или vim - это легкодоступные варианты. Просмотрите файл и его содержимое. Обратите внимание, что это обычный текстовый файл.

#### Скопируйте файл в свой домашний каталог HDFS. Назовите новый файл alice.txt

[student@localhost ~]$ hdfs dfs -put \

alice\_in\_wonderland.txt alice.txt

### 

### Убедитесь, что файл был загружен правильно. Мы будем использовать командную строку hdfs.

[student@localhost ~]$ hdfs dfs -ls

[student@localhost ~]$ hdfs dfs -cat alice.txt

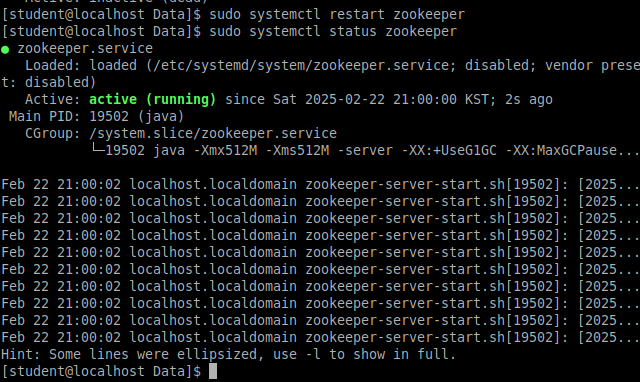
### 

### Теперь воспользуемся интерфейсом браузера файлов Hue

#### Убедитесь, что Zookeeper запущен. Если нет, запустите Zookeeper.

[student@localhost ~]$ sudo systemctl status zookeeper

Если zookeeper не запущен или произошел сбой, вы увидите экран, похожий на приведенный ниже



###### 

Если zookeeper работает правильно, вы увидите экран, похожий на приведенный ниже.

###### 

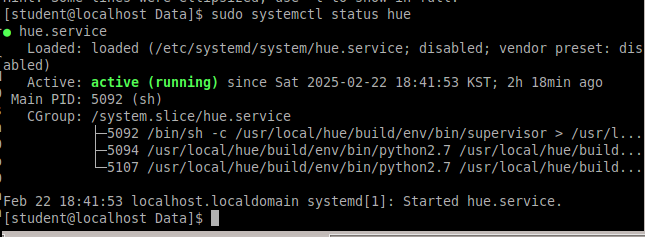
При необходимости перезапустите Zookeeper:

[student@localhost ~]$ sudo systemctl restart zookeeper

### Выполните аналогичный шаг, описанный выше, для Hue.

[student@localhost ~]$ sudo systemctl status hue

Как только Hue заработает должным образом, после проверки его состояния появится экран, похожий на приведенный ниже



### Откройте Hue из браузера Firefox. Используйте доступную в браузере закладку или используйте следующий URL-адрес: <http://localhost:8888>. Используйте следующую информацию для авторизации **Имя пользователя**: student **Пароль**: student

### Выберите из меню левой вкладки и перейдите на панель браузера файлов.

###### 

### Выберите файл alice.txt. Hue отобразит содержимое файла.

###### 

Если файл представляет собой текстовый файл, включая файлы JSON, CSV или обычный текст, Hue может отображать содержимое. Если файл находится в двоичном формате, таком как формат файла Parquet, используйте инструмент командной строки parquet-tools.

## Запустите свои первые преобразования Spark и сохраните результаты в HDFS

### Запустите PySpark с терминала

### Прочитайте alice.txt и проверьте его содержание

#### Создайте aliceRDD, прочитав alice.txt файл из вашего домашнего каталога HDFS

In [ ]: aliceRDD = sc.textFile("alice.txt")

#### Отобразите 5 строк aliceRDD. Используйте оператор действия take(n), чтобы вернуть n строк из aliceRDD. Используйте цикл for для перебора каждого элемента строки и, наконец, используйте print() для отображения содержимого каждой строки.

In [ ]: for line in aliceRDD.take(5):

print(line)



### Преобразуйте alice.txt, выбирая только те строки, в которых есть слово “rabbit”.

#### Создайте новый RDD со строками, содержащими слово "rabbit". Не делайте различий между прописными и строчными буквами, считаем, что регистр не важен. Сначала преобразуйте все строки в нижний регистр, а затем отфильтруйте строки, содержащие "rabbit".

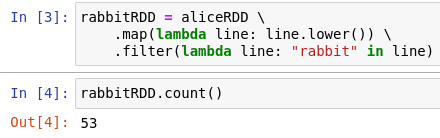
In [ ]: rabbitRDD = aliceRDD \

.map(lambda line: line.lower()) \

.filter(lambda line: "rabbit" in line)

#### Выведите 5 строк результирующего rabbitRDD и убедитесь, что все они содержат "rabbit".

#### Используйте действие count(), чтобы подсчитать, сколько строк содержит слово "rabbit". Сколько строк вы получили?



### Сохраните rabbitRDD в виде текстового файла в домашнем каталоге HDFS и убедитесь, что он был сохранен правильно. Назовите файл "rabbit.txt ".

#### Используйте действие saveAsTextFile(<path>) для сохранения rabbitRDD

In [ ]: rabbitRDD.saveAsTextFile("rabbit.txt")

#### Используйте командную строку HDFS (можно открыть второе окно терминала), чтобы убедиться, что rabbit.txt был сохранен. Обратите внимание, что rabbit.txt это каталог, а не текстовый файл. Это связано с тем, что Apache Spark — это распределенная параллельная система с несколькими исполнителями, обрабатывающими данные. Каждый исполнитель сохраняет свой раздел в указанном выходном пути.

###### 

#### Используйте параметр подкоманды -cat для просмотра содержимого part-00000

[student@localhost ~]$ hdfs dfs -cat rabbit.txt/part-00000

### 

### Убедитесь, что все строки содержат слово "rabbit", а весь текст написан строчными буквами.

#### Теперь используйте HUE, чтобы сделать то же самое.



## Объединение результатов HDFS в локальный файл

### Иногда может быть целесообразно объединить выходные результаты, разделенные на несколько файлов, в один локальный файл. Используйте командную строку HDFS с параметром --getmerge для создания единого файла

[student@localhost ~]$ hdfs dfs -getmerge rabbit.txt rabbit.txt

### 

### Убедитесь, что локальный файл с объединенным содержимым был сохранен на локальном диске

[student@localhost ~]$ ls

[student@localhost ~]$ cat rabbit.txt

# 

# Основы Python

## Работа с Jupyter

### Отредактируйте файл .bashrc и перейдите в режим Jupyter

###### 

### Запустите новый терминал или выполните файл .bashrc с помощью команды source

### Запустите оболочку PySpark. Оболочка должна автоматически запустить Firefox и запустить Jupyter. В случае, если браузер не запускается автоматически, запустите его по ссылке, указанной в командной строке.

### Создайте новый Notebook и выберите Python 3 (ipykernel)

###### 

### В Jupyter введите команду в ячейку, нажмите "Shift-Enter", чтобы выполнить команду и перейти к следующей ячейке. Протестируйте интерфейс, проверив наличие экземпляра объекта SparkContext и SparkSession.

###### 



## Применение функций Python к нашим данным

### Просмотрите kv1.txt в каталоге /home/student/Data. Создайте RDD из файла. Просмотрите содержимое RDD, чтобы убедиться, что данные были правильно загружены.

#### Перейдите в /home/student/Data и просмотрите kv1.txt. Вы можете использовать любой инструмент на ваш выбор для просмотра содержимого файла. Например, вы можете использовать KWrite, или команду Linux cat <filename> и т.д.

#### Используйте readTextFile() с локальным файлом. Путь к локальному файлу должен содержать полный URL-адрес, например file:/home/student/Data/kv1.txt

#### Используйте действие take(n), чтобы просмотреть содержимое файла. Ваш результат должен быть похож на приведенный ниже

###### 

### Создайте список из 2 элементов для каждой строки RDD

#### Используйте map() для преобразования каждой строки. Используйте split(<separator>), чтобы разделить строку, используя "\x01" в качестве разделителя.

#### Используйте take(n), чтобы убедиться, что у вас есть список для каждой строки в RDD

###### 

### Создайте функцию Python, которая принимает строку и заменяет строку по шаблону на строку для замены. Примените функцию ко второму элементу в списке.

#### Используйте def для создания функции. Используйте метод replace(<string>,<pattern>,<replacement>). Замените "val\_" на "value is " для второго элемента в списке

#### Используйте map() и примените вышеуказанную функцию для преобразования RDD.

#### Используйте take(n) для проверки вашего RDD. Ваши результаты должны быть похожи на приведенные ниже

###### 

### Создайте функцию Python, которая будет разрезать строку и возвращать только ее часть. Примените функцию для удаления префикса "value is " в строке. Вы можете создать именованную функцию и применить ее или использовать лямбда-нотацию с анонимной функцией.

#### Чтобы вырезать строку, используйте синтаксис [начальный\_индекс:конечный\_индекс\_не\_включительно]. Если конечный индекс оставить пустым, Python начнет с начального индекса и включит остальную часть строки в свой вывод. "value is " занимает индексы от 0 до 8. Мы хотим разрезать строку от индекса 9 до конца.

#### Используйте take(n) для проверки вашего RDD. Ваши результаты должны быть похожи на приведенные ниже.

###### 

### Каждая строка в текущем RDD представляет собой число, представленное в строковом типе данных. Преобразуйте каждую строку в кортеж. Кортеж будет состоять из двух элементов в каждом. Первым элементом будет целочисленное представление номера строки, а вторым элементом будет номер строки.

#### Используйте оператор (<первый элемент>, <второй элемент>) для создания кортежа. Используйте функцию int(< строка>) для преобразования строки в целое число.

#### Используйте take(n) для проверки вашего RDD. Ваши результаты должны быть похожи на приведенные ниже.

###### 

### Создайте функцию Python, которая принимает число и возвращает значение True, если это четное число. Используйте функцию для фильтрации RDD.

#### Используйте оператор %, чтобы проверить, является ли число четным. Оператор % возвращает остаток от деления. После деления числа на 2, если остаток равен 0, число является четным числом.

#### Используйте оператор if, чтобы вернуть либо True, если четное число, либо False, если число нечетное.

#### Используйте .filter(<Логическая функция>) для выбора строк, в которых первый элемент кортежа является четным числом.

#### Используйте take(n) для проверки вашего RDD. Ваши результаты должны быть похожи на приведенные ниже.

###### 

### В Python каждый тип данных имеет значение True или False. Для целых чисел каждое целое число, отличное от 0, является истинным. Единственное целое число, которое является ложным, равно 0. Повторите вышеописанное, используя вместо этого короткую лямбда-функцию.

#### Используйте оператор % с логическим оператором not.

#### Используйте take(n) для проверки вашего RDD. Ваши результаты должны быть похожи на приведенные ниже.

###### 

### Создайте функцию, которая перемещается по коллекции кортежей. Для каждого элемента кортежа, если элемент имеет целочисленный тип, добавьте к нему 1000 и выведите его. Если элемент имеет строковый тип, выведите его две копии. Если тип данных не является ни тем, ни другим, выведите " ERROR".

#### Используйте цикл for для перехода к каждому элементу в коллекции

#### Используйте вложенный цикл for для перехода к каждому элементу в кортеже

#### Используйте type(<variable>) чтобы определить тип переменной. Чтобы проверить наличие целого числа, используйте оператор is с int. Чтобы проверить тип строки, используйте оператор is с string.

### Получите первые 5 строк evenRDD, созданного выше. Передайте эту коллекцию функции, созданной на шаге выше в 2.8.

#### Вывод вашего результата должен выглядеть примерно так, как показано ниже.

###### 

## Отработка основ Python

### Создайте RDD, прочитав файл alice\_in\_wonderland.txt

### Создайте функцию, которая преобразует все слова в предложении так, чтобы они начинались с верхнего регистра. Это отличается от преобразования всего слова в верхний регистр

#### Создайте пустую строку: capWords = "". Поскольку каждое слово пишется с заглавной буквы, используйте оператор +, чтобы объединить новое слово с заглавными буквами в capWords.

#### Используйте метод split(<separator>) для разделения слов в строке

#### Используйте цикл for для перебора каждого слова

#### Для каждого слова добавляйте заглавную первую букву с помощью метода string upper()

#### Используйте срезы строк, чтобы добавить остальную часть слова. Подсказка: word(1:) - Если второй параметр пуст, предполагается конец строки и возвращается остальная часть строки

###### 

### Преобразуйте все слова в " alice\_in\_wonderland " в слова с заглавной буквы

#### Используйте преобразование map с помощью функции, которую вы создали выше

###### 

### Как оказывается, в Python есть хороший строковый метод, называемый capitalize(), который делает то же самое. Повторите шаги с 3.2 используя метод capitalize. Повторите шаг 3.3 используя свою новую функцию.

### Теперь сделаем обратную созданию заглавной буквы операцию. В этот раз мы пишем заглавными буквами все буквы, кроме первой.

# Работа с основными преобразованиями API

В этой лабораторной работе мы будем практиковаться в работе с различными преобразованиями основного API, которые были представлены в лекции. Запустите новую оболочку PySpark, работающую на Jupyter, чтобы начать выполнение лабораторной работы.

## Создание RDD из исходных файлов

### Убедитесь, что Hue запущен и работает. Следуйте инструкциям, приведенным в первой части, шаг 2.3.

### Создание RDD из текстовых файлов

#### Убедитесь, что файл "alice.txt " существует в вашем домашнем каталоге HDFS. Он был создан на основе предыдущей лабораторной работе. Для этого используйте команду hdfs dfs -ls или Hue.

#### Используйте команду SparkContext.textFile(<path>) для чтения файла alice.txt. Назовите новый RDD aliceRDD.

#### Убедитесь, что aliceRDD был создан правильно, используя take(5)

###### 

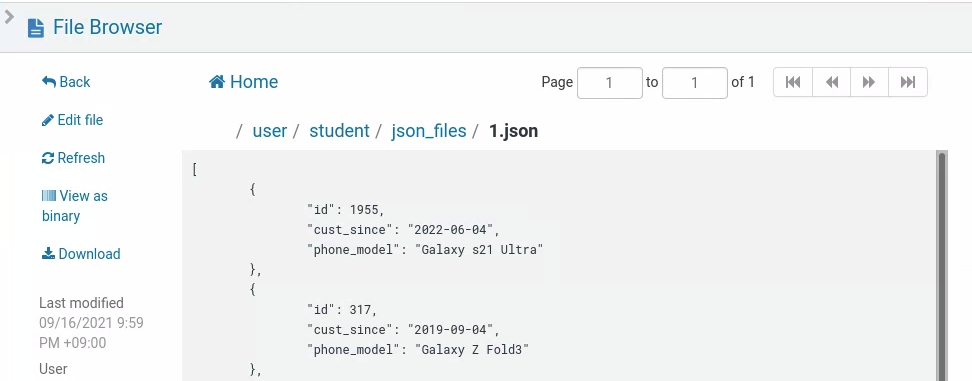
### Создание RDD из группы текстовых файлов

#### Перейдите в директорию /home/student/Data

#### Скопируйте json\_files директорию в домашнюю директорию HDFS

#### Убедитесь, что файлы были скопированы правильно. Используйте Hue для проверки и просмотра содержимого файла.

#### 



#### Из Jupyter используйте SparkContext.wholeTextFiles(<путь к исходным файлам>) для чтения каждого файла JSON как отдельного элемента во вновь созданном RDD. В поле путь к источнику вводите абсолютный путь. Нет необходимости указывать файловую систему, поскольку Spark в настоящее время настроен на чтение из HDFS по умолчанию. Полный путь можно увидеть из Hue. Это /user/student/json\_files. Назовите новый RDD как jsonRDD.

#### Убедитесь, что RDD был правильно создан с помощью take(1). Обратите внимание на формат элементов в RDD. Символ "\t" является табуляцией. Каждый элемент представляет собой парный кортеж, где первый элемент — это путь к файлу, а второй элемент - содержимое файла.

## Преобразования одного набора данных

### Рассмотрим flatMap преобразование

#### В новой ячейке импортируйте библиотеку для работы с JSON. Библиотека содержит методы, которые помогают читать файлы JSON.

import json

#### Используйте json.loads(<содержимое JSON>) чтобы извлечь запись. Из jsonRDD с помощью (<путь к файлу>, <содержимоеJSON>) формата. Используйте map() преобразование с json.loads(<содержимое JSON>) чтобы извлечь данные из JSON. В Python, доступ к каждому элементу кортежа имеет тот же синтаксис, что и доступ к элементам списка. Для доступа ко второму элементу кортежа, используйте tupleName[1]

#### Используйте take(2) для просмотра результата. Как json.loads() извлек содержимое JSON?

##### На предыдущем шаге, когда файлы JSON считывались с помощью wholeTextFiles(), каждый JSON файл превращался в JSON контент из (<путь>, <содержимое JSON>) кортежа, который был создан. Поскольку каждый файл содержит несколько записей JSON, когда json.loads() применяется к каждому содержимому JSON, многие записи JSON извлекаются как Python словари. Словарь Python содержит информацию в формате ключ:значение. Выходной формат выглядит как {key:value, key:value, key:value, …}. Несколько записей JSON внутри каждого содержимого JSON извлекаются как отдельные словари, и все записи возвращаются в виде списка.

###### 

#### Файл JSON был успешно разобран, но результат не в том формате, который необходим. Каждая запись JSON должна быть отдельным элементом. Обратите внимание на то, что выше мы запросили 2 строки с помощью take(2) и получили 2 списка, каждый из которых содержит несколько записей. Вместо этого используйте flatMap() преобразование чтобы создать новый RDD, где каждый элемент списка, созданного функцией преобразования, помещается в свою собственную строку. Модифицируйте код и используйте flatMap() вместо map().

#### Чтобы вернуть часть значения ключа:значения в словаре, используйте get("field name") метод. Так или иначе лучше будет использовать get("field name", None). Этот синтаксис возвращает None, что является эквивалентом Null в Python в случае, если словарь не содержит ключа запроса. Используйте map преобразование чтобы создать RDD в виде (<"id">, (<"cust\_since">,<"phone\_model">)). Результат представляет собой вложенный кортеж.

###### 

### Используйте distinct() чтобы удалить любой дублирующийся элемент в RDD

#### Начнем с aliceRDD, который был создан на предыдущем шаге. Используйте flatMap() преобразование вместе с str.split() методом чтобы создать RDD слов

#### Используйте действие RDD.count(), чтобы подсчитать количество слов.

#### Примените преобразование distinct().

#### Используйте RDD.count() на новом RDD, чтобы подсчитать количество уникальных слов в Alice In Wonderland.

###### 

## Работа с преобразованиями операторов набора

### Создадим RDD из коллекции.

#### Создайте список со следующими элементами и назовите его fruit1: ["Banana", "Pear", "Kiwi", "Peach", "Grape"]

#### Используйте SparkContext.parallelize(fruit1) чтобы создать fruit1RDD

#### Используйте действие collect(), чтобы проверить RDD

###### 

#### Создайте лист со следующими элементами и назовите его fruit2: ["Strawberry", "Kiwi", "Watermelon", "Banana", "Apple"]

#### Используйте SparkContext.parallelize(fruit2), чтобы создать fruit2RDD

#### Используйте действие collect(), чтобы проверить RDD

###### 

### Создайте unionRDD, используя преобразование dataset1.union(dataset2)

###### 

### Создайте intersectRDD, используя преобразование dataset1. intersection(dataset2)

###### 

### Создайте subtractRDD, используя преобразование dataset1. subtract(dataset2)

###### 

### Создайте cartesianRDD, используя преобразование dataset1. cartesian(dataset2)

###### 

## Работа с преобразованиями и действиями на основе разделов

### Используем mapPartitions() для увеличения производительности. mapPartitions() очень похож на map(), но имеет свои ключевые особенности. Когда мы используем mapPartitions(), Spark предоставляет итератор, при помощи которого мы можем итерироваться по строкам определенного раздела. Это позволяет разработчикам создавать функции, которые выполняют некоторые сложные операции, такие как создание подключения к базе данных только один раз для каждого раздела. Также итератор используется для отображения некоторого преобразования. Это может быть операция по обновлению или вставке записи в базу данных с использованием созданного соединения.

#### Создадим следующую функцию: Эта функция получает итератор в качестве своего входного параметра. Думайте об итераторе как о любом типе коллекции, которая содержит несколько элементов и имеет метод next(), который может использоваться циклом for для обхода коллекции. Функция печатает сообщение, а затем использует цикл for с итератором для получения каждого элемента. Оператор yield в Python — это простой способ создать новый итератор. В качестве альтернативы можно было бы создать пустой список и добавить каждый элемент в него.

def perPartition(it):

print("I just did a heavy operation once per partition")

for item in it:

yield "updated\_" + item

#### Создайте следующий список: my\_records = ["record1", "record2", "record3", "record4", "record5"]

#### Используйте SparkContext.parallelize(<collection>, <number of partitions>), чтобы создать RDD с 2 с двумя разделами, используя my\_records коллекцию, созданную выше. Назовите RDD, как recordsRDD.

#### Преобразуйте recordsRDD с mapPartitions(). Преобразование mapPartition(<func>) предоставляет итератор, который может использоваться функцией<func>. Используйте функцию perPartition() созданную выше для этих целей.

#### Используйте collect(), чтобы просмотреть результаты. Обратите внимание, что инструкция print была выполнена дважды для каждого раздела. Итератор использовался для обновления каждого элемента в каждом из разделов.

###### 

### Преобразование mapPartitionsWithIndex() очень похоже на mapPartitions(). Единственное реальное отличие заключается в том, что вместо просто итераторов также передается номер индекса текущего раздела. Сигнатурой для преобразования является mapPartitionsWithIndex(lambda index, iterator: <some function(index, iterator>)

#### Модифицируйте perPartition так, чтобы получать индекс и раздел. На этот раз измените инструкцию print, чтобы указать, какой раздел выполняет вывод на экран, используя переданный номер индекса раздела. Назовите функцию, perPartitionIndex.

#### Измените perPartitionIndex, чтобы обновленные записи включали номер индекса раздела.

#### Используйте collect(), чтобы просмотреть результат.

###### 

### Используйте действие foreachPartition(), чтобы вывести текст для каждого раздела

#### Создайте следующую функцию:

def actionPerPartition1(it):

print("I just did a foreachPartition action")

#### Используйте recordsRDD созданный выше с foreachPartition(<func>). Используйте actionPerPartition с <func> как входным параметром. Так как мы создали 2 раздела, actionPerPartition будет вызван дважды. Результат будет похож на следующий:

###### 

### Используйте действие foreachPartition(), чтобы вывести число строк в каждом разделе.

#### Иногда после преобразования данных один из разделов может быть перегружен большим количеством данных по сравнению с другими разделами. Это называется проблемой перекоса. Создайте следующую функцию, которая будет выводить количество элементов в каждом разделе.

def actionPerPartition2(it):

print("I have", len(list(it)), "elements")

#### Итератор, передаваемый foreachPartition, на самом деле является генератором. Генератор можно рассматривать как ленивый список. Список не актуализирован и на самом деле может продолжать расти. Ленивый список полезен для создания списка из потокового источника, например, когда конец списка неизвестен. Чтобы получить текущую длину ленивого итератора, мы приводим его в фактический список, а затем определяем длину списка.

#### Используйте foreachPartition() на recordsRDD с actionPerPartition2. Результат будет похож на следующий.

###### 

#### Обратите внимание, что выходные данные включают в себя выходные данные функции perPartition(), созданной выше. Когда ленивый итератор актуализируется путем приведения в список, Spark следует своей линии зависимостей, чтобы получить фактический список. Вдоль этой линии зависимостей была проведена трансформация mapPartitions(perPartition). Это заставляет Spark выполнять это преобразование по мере необходимости.

#### В этот раз, используйте foreachPartition на records2RDD с actionPerPartiton2. Чем результаты отличаются? Что происходит?

##### На самом деле происходит то же самое, records2RDD зависимости отличаются от recordsRDD зависимостей. records2RDD имеет mapPartitionWithIndex(perPartionIndex) в своей ветке зависимостей.

###### 

## Преобразования, изменяющие количество разделов RDD

### Используйте coalesce(), чтобы уменьшить количество разделов.

#### Определите функцию printElements, как показано далее:

def printElements(iterator):

for item in iterator: print(item)

print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

#### Создайте RDD состоящий из чисел от 1 до 9. Создайте RDD с 4 разделами.

#### Используйте getNumPartitions(), чтобы вывести количество разделов.

#### Используйте foreachPartition с printElements на RDD, чтобы вывести элементы в каждом разделе

#### Используйте coalesce(), чтобы уменьшить количество разделов до 2

#### Используйте getNumPartitions() еще раз, чтобы напечатать новое количество разделов.

#### Используйте foreachPartition с printElements на новом RDD для печати элементов в каждом разделе

### Используйте repartition(), чтобы изменить количество разделов

#### Создайте RDD состоящий из номеров от 1 до 9. Создайте RDD с 4 разделами.

#### Используйте getNumPartitions(), чтобы вывести количество разделов.

#### Используйте foreachPartition с printElements на RDD, чтобы вывести элементы в каждом разделе

#### Используйте repartition(4) на RDD. Обычно перераспределение используется для изменения количества разделов и одновременной перетасовки данных. Сохраняя количество разделов неизменным, легче наблюдать за перетасовкой данных при преобразовании.

#### Используйте getNumPartitions(), чтобы вывести количество разделов.

#### Используйте foreachPartition с printElements на новом RDD, чтобы вывести элементы в каждом разделе

###### 

###### 



## Различные преобразования

### Используйте преобразование sample() для отбора данных из RDD и создайте частичный RDD. Это преобразование очень полезно при исследовании данных в большом наборе данных. sample() может использоваться с повторами или без них. Если установлено значение True, выборочные данные могут повторяться. Если установлено значение False, то после выборки точки данных она не заменяется другой, и, следовательно, данные не повторяются

#### Создайте RDD состоящий из чисел от 0 до 100. Используйте в Python функцию range(100) чтобы создать числа. Используйте SparkContext.parallellize, чтобы создать RDD. Назовите RDD как 100RDD.

#### Используйте sample(<with Replacement?>,<fraction to sample>,<seed>), чтобы отобрать 100RDD. Выберите 20% (0.2) данных без замены и используйте начальное значение по вашему выбору.

#### Повторите описанное выше, но на этот раз сгенерируйте случайное число для начального значения. Используйте следующий код для генерации случайного числа.

import random

seed = int(random.random())

#### На этот раз возьмите выборку с заменой. Обратите внимание, что, если для параметра «with Replacement» установлено значение True, выбранные данные могут повторяться.

###### 