# mrjob

mrjob to Pythonowa biblioteka MapReduce, stworzona przez Yelp, która opakowuje strumieniowanie Hadoop, umożliwiając pisanie aplikacji MapReduce w sposób bardziej Pythonowy. mrjob umożliwia pisanie wieloetapowych zadań MapReduce w czystym Pythonie. Zadania MapReduce napisane za pomocą mrjob można testować lokalnie, uruchamiać w klastrze Hadoop lub uruchamiać w chmurze przy użyciu Amazon Elastic MapReduce (EMR).

Pisanie aplikacji MapReduce za pomocą mrjob ma wiele zalet:

* mrjob jest cały czas rozwijaną biblioteką i działa z Pythonem 3.
* mrjob posiada obszerną (i dobrze napisaną), dokumentację, jest lepiej udokumentowana niż jakikolwiek inny framework lub biblioteka obsługująca Python na Hadoopie.
* aplikacje mrjob można uruchamiać i testować bez instalowania usługi Hadoop, co umożliwia tworzenie i testowanie przed ich wdrożeniem w klastrze Hadoop.
* mrjob umożliwia pisanie aplikacji MapReduce w jednej klasie, zamiast pisania oddzielnych programów dla mappera i reduktora.

Chociaż mrjob jest świetnym rozwiązaniem, ma swoje wady. mrjob jest uproszczona, więc nie zapewnia takiego samego poziomu dostępu do Hadoop, jaki oferują inne interfejsy API. mrjob nie używa typedbytes, więc inne biblioteki mogą być szybsze.

## Instalacja

Instalacja mrjob jest prosta; można ją zainstalować za pomocą pip za pomocą następującego polecenia:

$ pip install mrjob

Lub bezpośrednio ze źródła (za pomocą polecenia git clone):

$ python setup.py install

## WordCount w mrjob

Liczenie słów z pomocą map reduce jest popularnym rozwiązaniem, którego przykład widzieliśmy już bezpośrednio w Pythonie z zastosowaniem dwóch skryptów do każdego z zadań. Tym razem zobaczymy, jak można to wszystko zrobić jednym skryptem z zastosowaniem mrjob. Przykład 2-3 wykorzystuje mrjob do implementacji algorytmu WordCount.

##### Przykład 2-3. python/MapReduce/mrjob/word\_count.py

from mrjob.job import MRJob

**# definijemy klasę, która dziedziczy po klasie MRJob i dzięki temu mamy dostęp do jej gotowych metod: map, reduce i combine.**

class MRWordCount(MRJob):

**# funkcja mapper przyjmuje dwie wartości: klucz, wartość. Tutaj ignorujemy pierwszy argument (klucz) i przyjmujemy tylko wartość - line**

def mapper(self, \_, line):

for word in line.split():

yield(word, 1)

**# funkcja mapper przyjmuje dwie wartości: klucz, wartość. Tutaj zwraca klucz – word i sumę jego wartości.**

def reducer(self, word, counts):

yield(word, sum(counts))

if \_\_name\_\_ == ‘\_\_main\_\_’:

MRWordCount.run()

Aby uruchomić mrjob lokalnie, potrzeba jedynie pliku tekstowego do wczytania. Aby uruchomić zadanie lokalnie i policzyć częstotliwość słów w pliku o nazwie input.txt, użyj następującego polecenia:

$ python word\_count.py input.txt

Wynik zależy od zawartości pliku wejściowego, ale powinien wyglądać podobnie do przykładu 2-4.

##### Przykład 2-4. Wynik word\_count.py

“be” 2

“jack” 2

“nimble” 1

“quick” 1

## Jak to działa?

Zadanie MapReduce jest zdefiniowane jako klasa MRWordCount. W bibliotece mrjob klasa dziedzicząca po MRJob zawiera metody definiujące kroki zadania MapReduce. Kroki w aplikacji mrjob to mapper, combiner i reducer. Klasa dziedzicząca MRJob musi tylko zdefiniować jeden z tych kroków.

Metoda mapper()  definiuje program mapujący dla zadania MapReduce. Pobiera klucz i wartość jako argumenty i zwraca krotki (klucz\_wyjściowy, wartość\_wyjściową). W przykładzie WordCount (Przykład 2-4), program mapper zignorował klucz wejściowy i podzielił wartość wejściową, aby uzyskać słowa i liczby.

Metoda combiner() definiuje moduł łączenia dla zadania MapReduce. Combiner to proces, który przebiega po mapperze, a przed reducerem. Jako dane wejściowe odbiera wszystkie dane emitowane przez mapper, a dane wyjściowe combinera są wysyłane do reducera. Wejściem combinera jest klucz, który został przekazany przez mapper, oraz wartość, która jest generatorem, który zwraca wszystkie wartości otrzymane przez jeden mapper, który odpowiada danemu kluczowi. Combiner daje (yields) krotki (klucz\_wyjściowy, wartość\_wyjściową) jako dane wyjściowe.

Metoda reducer()  definiuje reduktor dla zadania MapReduce. Pobiera klucz i iterator wartości jako argumenty i daje (yields) krotki (klucz\_wyjściowy, wartość\_wyjściową). W przykładzie 2-4 reduktor sumuje wartość dla każdego klucza, który reprezentuje częstotliwość słów w danych wejściowych.

Ostatnim składnikiem zadania MapReduce napisanego za pomocą biblioteki mrjob są dwie linie na końcu pliku:

if \_\_name\_\_ == ‘\_\_main\_\_’:

MRWordCount.run()

Te linie umożliwiają wykonanie mrjob; bez nich aplikacja nie będzie działać.

## Uruchamianie mrjob

Uruchamianie aplikacji MapReduce za pomocą mrjob jest podobne do wykonywania dowolnego innego programu w Pythonie. Wiersz poleceń musi zawierać nazwę aplikacji mrjob i plik wejściowy:

$ python mr\_job.py input.txt

Domyślnie mrjob zapisuje dane wyjściowe na standardowe wyjście (stdout).

Wiele plików można przekazać do mrjob jako dane wejściowe, określając nazwy plików w wierszu poleceń:

$ python mr\_job.py input1.txt input2.txt input3.txt

mrjob może również obsługiwać dane wejściowe przez standardowe wejście (stdin):

$ python mr\_job.py < input.txt

Domyślnie mrjob działa lokalnie, umożliwiając tworzenie i debugowanie kodu przed przesłaniem go do klastra Hadoop.

Aby zmienić sposób uruchamiania zadania, dodaj opcję -r/--runner. Tabela 2-2 zawiera opis prawidłowych wyborów dla opcji runner:

|  |  |
| --- | --- |
| -r inline | (Default) Run in a single Python process |
| -r local | Run locally in a few subprocesses simulating some Hadoop features |
| -r hadoop | Run on a Hadoop cluster |
| -r emr | Run on Amazon Elastic Map Reduce (EMR) |
| Tabela 2-2. Opcje mrjob runner | |

Użycie opcji runner umożliwia uruchamianie programu mrjob w klastrze Hadoop z danymi wejściowymi określonymi z HDFS:

$ python mr\_job.py -r hadoop hdfs://input/input.txt

mrjob umożliwia również uruchamianie aplikacji w EMR bezpośrednio z wiersza poleceń:

$ python mr\_job.py -r emr s3://input-bucket/input.txt

## Najpopularniejsze słowa w tekście

Przykład 2-5 pokazuje jak znaleźć najczęściej występujące słowa.

##### Przykład 2-5.  most\_used\_words.py

from mrjob.job import MRJob

from mrjob.step import MRStep

import re

WORD\_RE = re.compile(r"[\w']+") # regex do znajdywania słów

class MRMostUsedWord(MRJob):

    def steps(self):

        return [

            MRStep(mapper=self.mapper\_get\_words,

                   combiner=self.combiner\_count\_words,

                   reducer=self.reducer\_count\_words),

            MRStep(reducer=self.reducer\_find\_max\_word)

        ]

    def mapper\_get\_words(self, \_, line):

        # zwróć każde słowo z linii

        for word in WORD\_RE.findall(line):

            yield (word.lower(), 1)

    def combiner\_count\_words(self, word, counts):

        # optymalizacja: zsumuj przejrzane do tej pory słowa

        yield (word, sum(counts))

    def reducer\_count\_words(self, word, counts):

        # przekaż wszystkio (counts, word) pary do tego samego reducera.

        # taka forma counts w parze z word pozwala nam na wykorzystanie funckji Pythonowej max().

        # Jako klucz zwracamy None -> taki output ułatwia input do reducera w kolejnym kroku

        yield None, (sum(counts), word)

    # zignoruj klucz

    def reducer\_find\_max\_word(self, \_, word\_count\_pairs):

        # każdy element word\_count\_pairs jest krotką (count, word),

        # dlatego yielding jednego zwrócio key=counts, value=word

        yield max(word\_count\_pairs)

if \_\_name\_\_ == '\_\_main\_\_':

    MRMostUsedWord.run()

W celu uruchomienia programu na Hadoopie można skorzystać z polecenia:

$ python most\_used\_words.py -r hadoop hdfs:///user/hduser/input/poem.txt