Algorytmy skalowalnego przetwarzania danych — projekt 1

dr Piotr Przymus, mgr Mikołaj Fejzer, dr Krzysztof Rykaczewski

3 marca 2020

Spis treści

1 Teoria oraz definicje

1.1 Definicje funkcji

Zgodnie z publikacją *MapReduce: Simplified Data Processing on Large Clusters*, Jeffrey Dean and Sanjay Ghemawat, Google, Inc., 2004, koncepcje funkcji map oraz `reduce` zostały zaczerpnięte z programowania funkcyjnego.

1.1.1 Funkcja `map`

To przekształcenie wejścia na postać, którą da się agregować

```
map(in_key, in_value) -> list(out_key, intermediate_value)
```

Uwaga: w poniższych zadaniach pomijamy `in_key`, więc jako sygnaturę tej funkcji równie dobrze moglibyśmy przyjąć

```
map(in_value) -> list(out_key, intermediate_value)
```

1.1.2 Funkcja `reduce`

To właściwa agregacja do listy wyników

```
reduce(out_key, list(intermediate_value)) -> list(out_value)
```

1.2 Definicje mechanizmów w implementacji silnika algorytmu MapReduce (takiej jak Apache Hadoop):

- **Zadanie Map** – wykonuje funkcję `map` na lokalnych danych.
- **Zadanie Reduce** – wykonuje funkcję `reduce` na lokalnych danych.
- **Krok Map** – wykonuje **zadanie Map** równolegle na każdej z maszyn, nadzorca uruchamia powtórnie zadania przerwane.
- **Krok Sort & Shuffle** – wykonuje sortowanie oraz migrację danych pomiędzy maszynami, przez rozproszony system plików.
- **Krok Reduce** – wykonuje **zadanie Reduce** na każdej z maszyn, nadzorca uruchamia powtórnie zadania przerwane.
- **Algorytm MapReduce** – wykonanie w kolejności wczytania danych, **kroku Map**, **kroku Sort & Shuffle**, **kroku Reduce** oraz zapisanie wyników.

1.3 Uwagi

- W ogólności operacja `reduce` wystarczy, że jest łączna i ma element neutralny. Natomiast gdyby shuffle nie działał deterministycznie (np. zwraca elementy w losowej kolejności, tak jak wyliczył), to musi być też przemienna.
- MapReduce nie musi być wcale szybki. Zależy to od tego jak dobrze da się rozproszyć zadania oraz ile ich jest. Może sie to przełożyć na koszt w komunikacji i na etap Sort & Shuffle. Zaletą MapReduce'a jest natomiast to, że mamy wygodny framework, a przez to martwimy się tylko o napisanie dwóch funkcji: `mapper` i `reducer`.

2 Hashowanie

2.1 Na czym polega?

- Opis, przykłady, funkcje,
- Skleja podobne elementy.

- Jest to funkcja w jedną stronę.
- Ma tę własność, że na pewno te same klucze wpadną do tego samego *bucket*-a (kubełka).
- Na odwrót, jeśli coś wpadło do różnych kubełków to nie mogły być te same klucze!
- Ponadto dobra funkcja hashująca równo dzieli dziedzinę, przez co mam podział danych na podobne wielkością zbiory. Zachowuje warstwy/strukturę.
- Zastępuje nam pseudolosowe rozdzielenie danych, ale według warstw.

Zobaczcie, że te same klucze nie idą do innych bucket-ów, gdyż podział jest deterministyczny.

Rozważmy taki podział na równomierne grupy: bierzemy do ręki element i losujemy liczbę całkowitą z przedziału $[0, N-1]$. Wówczas elementom zostaną przydzielone zasadniczo równomierne pewne etykiety. Jaki jest problem? Ten sam klucz/element może dostać dwie różne etykiety. Jak zrobić, żeby te same elementy dostały tą samą etykietę, a przy okazji podział był równomierny?

Wprowadzenie elementu losowo-deterministycznego. Deterministycznego, bo te same elementy zawsze trafią do tych samych kubełków (nigdy innych), a losowego, bo to jakby dla każdego elementu losować numer kubełka i tam powinien trafić ten element.

2.2 Przykład (WordCount)

Załóżmy, że mamy jakąś funkcję zamieniającą łańcuch znaków na liczbę całkowitą. Np.

$a - 2^0 = 1$

$b - 2^1 = 2$

$c - 2^2 = 4$

etc.

Załóżmy, że mamy do dyspozycji 5 węzłów reduce. Wówczas zadania:

```
(w1, [1,1,1,1])
(w2, [1])
(w3, [1,1,1])
(w4, [1,1])
...
```

można równomiernie rozdzielić między węzły poprzez zastosowanie funkcji hashującej $h(w) \mod 5$. Np.

```
h(abc) (mod 5) = 7 (mod 5) = 2
h(aa) (mod 5) = 2 (mod 5) = 2
h(bc) (mod 5) = 6 (mod 5) = 1
h(c) (mod 5) = 4 (mod 5) = 4
h(b) (mod 5) = 2 (mod 5) = 2
```

```
h(a) (mod 5) = 1 (mod 5) = 1
h(c) (mod 5) = 4 (mod 5) = 4
h(ab) (mod 5) = 3 (mod 5) = 3
h(ac) (mod 5) = 5 (mod 5) = 0
h(bb) (mod 5) = 4 (mod 5) = 4
h(cc) (mod 5) = 8 (mod 5) = 3
```

Stąd mamy

node	key	**node**	key	**node**	key
0	ac	**1**	a, bc	**2**	b, aa, abc
3	ab, cc	**4**	c, bb		

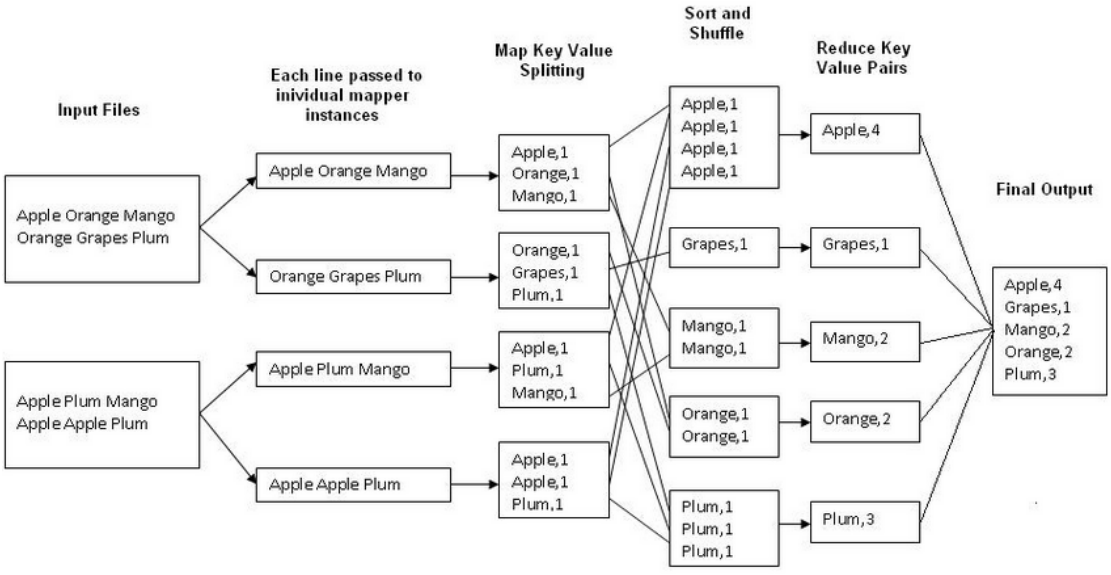
W miarę równomiernie. Przy dużej liczbie przykładów grupy się wyrównają.

Powoduje to zrównoważenie obciążenia (*load balancing*) na węzłach obliczeniowych.

3 Zadania

3.1 Word Count: pierwsze podejście (20 pkt)

Czyli *Hello world* dla MapReduce. Celem jest policzenie częstotliwości występowania poszczególnych słów w dużym zbiorze tekstów.



Rysunek 1: Word Count[1]

Należy przygotować dwa programy *Mapper* i *Reducer* (dowolny język programowania), programy mają czytać ze standardowego wejścia i wypisywać wyniki na standardowe wyjście. W tym przykładzie będziemy wykorzystywali uproszczony lokalny framework i *Spark Pipe*, stąd specyfikacja działania **zadań Reduce**.

3.1.1 Mapper

Czyta linie ze standardowego wejścia, dzieli na słowa (separatorem są białe znaki: spacja, tabulacja, etc.). Program zwraca mapowania `klucz\twartosc\n`. W omawianym przypadku dla każdego słowa wypisuje na standardowe wyjście `slowo\t1\n`.

Uwaga: Na początku najwygodniej będzie nam pracować na jakimś przykładowym pliku (stworzonym samemu).

3.1.2 Reducer

Czyta ze standardowego wejścia; linie są w formacie wyjścia *Mapper*-a.

Program *Reducer* otrzymuje na wejściu jeden lub więcej kluczy wraz z wartościami.

```
k1 : [v_11, v_12, ..., v_1n],
k2 : [v_21, v_22, ..., v_1m],
...
```

W naszym przypadku korzystamy z *Hadoop streaming*, dlatego format, który otrzyma program Reduce na wejściu będzie taki:

```
k1\tv_11
k1\tv_12
...
```

Jeden klucz może być przetwarzany tylko przez jedno zadanie Reduce, ale to samo zadanie Reduce może przetwarzać wiele kluczy. Dlatego w tym formacie, gdy w nowej linii pojawi się inny klucz, znaczy to, że skończyliśmy przetwarzanie dla danego klucza i zaczynamy przetwarzać nowy klucz.

3.2 Ulepszony Mapper (20 pkt)

3.2.1 Mapper I: stopwords, interpunkcja, wielkośś liter

Wersja rozszerzona punktu 1.1. Zadania Map tak jak w wersji podstawowej, tylko dodatkowo czyszczą one tekst ze znaków interpunkcyjnych oraz ze *stopwords* (najczęściej występujące słowa języka). Należy również znormalizować wielkość liter w tekście.

Uwagi

[1] Źródło: http://javax4u.blogspot.com/2012/11/hadoop.html

- Wyrazy podlegają prawu Zipfa (the, 1), (a, 1) etc. występowały by nadzwyczaj często.
- stopwords trzeba umieścić w odpowiedniej strukturze. Uwaga na złożoności w zależności od doboru struktury danych i jej implementacji ($O(n)$, $O(log(n))$, $O(1)$).

3.2.2 Mapper II: lematyzacja

Wersja rozszerzona 3.2.1 tylko dodatkowo przekształca słowa do postaci bazowej z wykorzystaniem lematyzera (np. z pakietu NLTK, lub innych).

3.3 Uproszczony framework MapReduce (20 pkt)

Przygotować kod sklejający wywołania programów Mapper i Reducer tak, aby uzyskać oczekiwany wynik. Musimy zapewnić krok wczytujący dane z plików oraz krok pośredni (sort & shuffle) i złożyć z działaniem przygotowanych programów Mapper i Reducer. W wersji podstawowej zakładamy, że jest tylko jedno zadanie Map i jedno zadanie Reduce. Dlatego krok sort & shuffle ogranicza się do sortowania.

3.3.1 Uwagi

- `sort` w `bash` to nie mu być ten sam `sort` co w `hadoop`, dlatego wyniki mogą być różne w zależności czy `hadoop` klucz potraktuje leksykograficznie czy numerycznie. Z tego powodu czasem warto klucz nazwać `(1, 2)` lub `1%2`, zamiast po prostu `1 2`.
- Ustawienie lokali `LC_ALL=C` powinno sprawić, że sortowanie w `bash`-u i `sparku` będzie takie samo.
  - What does `LC_ALL=C` do?

3.4 Word Count w Spark: pierwsze podejście (20 pkt)

3.4.1 Instalacja i konfiguracja Hadoop w trybie pojedynczej instancji

1. Upewnić się, że posiadamy aktualne Java SDK.
2. Ściągnąć najnowszą wersję `sparka` (zintegrowana z hadoop).
3. Rozpakować.
4. Ustawić zmienne środowiskowe (zgodnie z własnym systemem).

```
# set to the root of your Java installation
export JAVA_HOME=/usr/java/latest
```

3.4.2 Word Count w Spark.pipe

Przygotować kod sklejający wywołania programów Mapper i Reducer tak, aby uzyskać oczekiwany wynik.

Poniżej zakładamy, że w katalogu `word_count`, znajdują się:

- word_count/maper - program wykonujący map
- word_count/reduceer - program wykonujący reduce
- argv[1] - plik z danymi, w tym przypadku książki z projektu Gutenberg

```
from __future__ import print_function

import sys
from operator import add

from pyspark.sql import SparkSession


if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: wordcount <file>", file=sys.stderr)
        exit(-1)

    spark = SparkSession\
        .builder\
        .appName("PythonWordCount")\
        .getOrCreate()

    lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0])
    countsR = lines.pipe("word_count/maper")\
        .sortBy(lambda line: line.split("\t")[0])\
        .pipe("word_count/reducer")

    for line in countsR.collect():
        print("%s" % (line))

    spark.stop()
```

Aby uruchomić używamy:

```
bin/spark-submit word_count.py plik_z_tekstem.txt
```

3.5 Word Count w Spark: różne wersje (20 pkt)

3.5.1 Word Count w Spark: przy użyciu flatmap

Popraw kod w miejscach oznaczonych TODO.

```
from __future__ import print_function

import sys
from operator import add

from pyspark.sql import SparkSession

def Map(r):
    # Linia tekstu w kluczu
    key, value = r[0], r[1]
    # TODO: dostosuj kod z poprzedniego zadania
    # return lista (klucz, wartosc)

def Reduce(r):
    key, value = r[0], r[1]
    # TODO: dostosuj kod z poprzedniego zadania
    # return lista (klucz, wartosc), wartosc to suma

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: wordcount <file>", file=sys.stderr)
        exit(-1)

    spark = SparkSession\
        .builder\
        .appName("PythonWordCount")\
        .getOrCreate()

    lines = spark.read.text(sys.argv[1]).rdd
    countsR = lines.flatMap(Map)\
                   .groupByKey()\
                   .map(Reduce)

    for result in countsR.collect():
        print("Key = %s,  value = %d" % (result[0], result[1]))

    spark.stop()
```

3.5.2 WordCount w Spark (przykład bazowy)

Uruchomić przykład word counta dostępny w przykładach sparka.

Dla pythona plik `examples/src/main/python/wordcount.py`, zawiera kod:

```python
from __future__ import print_function

import sys
from operator import add

from pyspark.sql import SparkSession


if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: wordcount <file>", file=sys.stderr)
        exit(-1)

    spark = SparkSession\
        .builder\
        .appName("PythonWordCount")\
        .getOrCreate()

    lines = spark.read.text(sys.argv[1]).rdd.map(lambda r: r[0]) # (k,v) -> k

    counts = lines.flatMap(lambda x: x.split(' ')) \ # linia na listę słów
                  .map(lambda x: (x, 1)) \ # słowo na (klucz, 1)
                  .reduceByKey(add) # redukcja wartości przy użyciu add

    # Zebranie wyników
    output = counts.collect()

    # Wyświetlenie wyników
    for (word, count) in output:
        print("%s: %i" % (word, count))

    spark.stop()
```