Pytanie 1

Częściowo poprawnie

Punkty: 1,00

Oflaguj pytanie

Programista aplikacji Spark operuje na obiekcie `DataFrame` o nazwie **`gameInfosDF`**, którego schemat jest następujący:

```
root
 |-- _c0: integer (nullable = true)
 |-- score_phrase: string (nullable = true)
 |-- title: string (nullable = true)
 |-- url: string (nullable = true)
 |-- platform: string (nullable = true)
 |-- score: double (nullable = true)
 |-- genre: string (nullable = true)
 |-- editors_choice: string (nullable = true)
 |-- release_year: integer (nullable = true)
 |-- release_month: integer (nullable = true)
 |-- release_day: integer (nullable = true)
```

Zaimplementował on operację na tym obiekcie z wykorzystaniem polecenia SQL,
tworząc w ten sposób wynikowy obiekt **`result`**.

```
gameInfosDF.createOrReplaceTempView("gameInfos")
val result = spark.sql("""
select platform, count(*) as ile
from gameInfos
group by platform
having count(*) > 100""")
```

Uzupełnij poniższe polecenie wykorzystujące metody obiektu DataFrame tak, aby obiekt `result2` utworzony za jego pomocą był pełnym odpowiednikiem obiektu **result**.

```
import org.apache.spark.sql.functions._
val result2 = gameInfosDF.select ✗ ($" platform ✓ ").
agg ✓ (count($"platform"). as ✓ ("ile")).
where ✓ ($"ile" > 100)
```

Pytanie 2

Częściowo poprawnie

Punkty: 1,00

Usuń flagę

Platformy Hive i Pig pozwalają definiować złożone przetwarzanie danych na platformie Hadoop, które wykonywane jest następnie przy użyciu jednego z silników (MapReduce, Tez, Spark).

Hive pozwala definiować to przetwarzanie za pomocą języka SQL, natomiast Pig za pomocą języka Pig Latin.

Polecenia języka Pig Latin bardzo często odpowiadają klazulom w języku SQL.

Dopasuj do siebie polecenia Pig Latin o odpowiadające im klauzule.

COGROUP	Brak odpowiednika ✓
GROUP	GROUP BY ✓
FOREACH	Brak odpowiednika ✗
HAVING	WHERE ✗
ORDER BY	ORDER BY ✓
FILTER	WHERE ✓

Pytanie 3

Niepoprawny(a)

Punkty: 1,00

Usuń flagę

Która z wymienionych cech **nie byłaby zaletą** narzędzia użytego do obsługi warstwy obsługującej (ang. *serving layer*) przechowującej obrazy wsadowe (ang. *batch view*) w systemie Big Data opartym o architekturę Lambda.

Wybierz jedną odpowiedź:

- a. Niemutowalność danych polegająca na braku możliwości zmiany wprowadzonych danych ✖
- b. Zdolność do partycjonowania pionowego
- c. Systemowy brak możliwości odczytu pojedynczych danych - wsparcie tylko dla odczytu całości
- d. Tolerancja na błędy sprzętowe

Pytanie 4

Częściowo poprawnie

Punkty: 1,00

Oflaguj pytanie

Za pomocą poniższego kodu programista Sparka utworzył zmienną **tabRDD** typu `RDD[(String,Int,Double)]`

```
var tab_source = new ListBuffer[(String, Int, Double)]()
for (a <- "abcde"; b <- (1 to 10); c <- (1 to 10))
    tab_source += ((a.toString,b, c.toDouble/10))
val tab_source_list = tab_source.toList
val tabRDD = sc.parallelize(tab_source_list)
```

Następnie wykonane zostały trzy kolejne polecenia. Określ po każdym z poleceń typ powstałej zmiennej oraz wartość metody `count()` (lub liczbę elementów składowych jeśli wynik, np. `Map`, nie posiada metody `count()`) wykonanej na rzecz tej zmiennej.

Polecenie 1:

```
val v = tabRDD.map(p => (p._3,p._2))
```

- Typ zmiennej v to: RDD ✔[(Double ✔, Int ✔)]
- Liczba elementów składowych (wynik metody `count()`): 500 ✔

Polecenie 2:

```
val v = tabRDD.take(50)
```

- Typ zmiennej v to: RDD ✖[(String ✔, Int ✔, Double ✔)]
- Liczba elementów składowych (wynik metody `count()`): 50 ✔

Polecenie 3:

```
val v = tabRDD.map(p => (p._3,p._2)).reduceByKey(_ + _)
```

- Typ zmiennej v to: RDD ✔[(Double ✔, Int ✔)]
- Liczba elementów składowych (wynik metody `count()`): 10 ✔

Pytanie 5

Poprawnie

Punkty: 1,00

Oflaguj pytanie

YARN funkcjonuje na platformie Hadoop jako manager zasobów. Każda aplikacja żąda określonych zasobów, a YARN w zależności od możliwości je przyznaje.

YARN obsługuje trzy typy mechanizmów szeregowania zadań:

- FIFO,
- Fair i
- Capacity.

Załóż że w naszym przypadku wykorzystywanym mechanizmem jest ***Fair*** . Aby określać ilości zasobów przyporządkowywane poszczególnym aplikacjom YARN wykorzystuje w takim przypadku metodę dominujących zasobów (***Dominant Resource Fairness***).

Załóż, że na naszej platformie do dyspozycji YARN są 2 typy zasobów CPU i RAM.
Klaster Hadoop składa się z:

- **1** - liczba węzłów master (na którym funkcjonuje tylko i wyłącznie *resource manager*)
- **4** - liczba węzłów roboczych, z których każdy dysponuje
  - **8 CPU**
  - **6 GB RAM**

Do YARN zgłosiły się 3 aplikacje (A, B, C) żądając następującej wielkości poszczególnych zasobów:

- Aplikacja **A: 20 CPU, 18 GB RAM**
- Aplikacja **B: 16 CPU, 8 GB RAM**
- Aplikacja **C: 12 CPU, 12 GB RAM**

Uwzględniając powyższe oblicz przydział zasobów dokonany przez YARN dla poszczególnych aplikacji, a następnie uzupełnij poniższe zdania.

- Zasobem dominującym dla aplikacji **A** jest:
  ○ CPU ◉ RAM ✔ ○ oba są tak samo dominujące
- Zasobem dominującym dla aplikacji **B** jest:
  ◉ CPU ✔ ○ RAM ○ oba są tak samo dominujące
- Zasobem dominującym dla aplikacji **C** jest:
  ○ CPU ◉ RAM ✔ ○ oba są tak samo dominujące

W wyniki alokacji zasobów następujące zasoby zostały **wykorzystane w 100%**:

☐ CPU ☑ RAM ✔

Zasoby przyznane poszczególnym aplikacjom wyglądają następująco:

- Aplikacja A: 10 ✔ CPU, 9 ✔ GB RAM
- Aplikacja B: 12 ✔ CPU, 6 ✔ GB RAM
- Aplikacja C: 9 ✔ CPU, 9 ✔ GB RAM

4x (8 CPU, 6GB RAM) - klaster

20a + 16b + 12c <= 32
18a + 8b + 12c <= 24
Dominujące zasoby:
A: CPU = 20/32 = 5/8; **RAM = 3/4**
B: **CPU = 1/2**; RAM = 1/3
C: CPU = 3/8; **RAM=1/2**

20a + 16*3a/2 + 12*3a/2 <= 32
20a + 24a + 18a <= 32
62a <= 32
a <= 32/62

18a + 8*3a/2 + 12*3a/2 <= 24
18a + 12a + 18a <= 24
48a <= 24
a <= 1/2

$\frac{3a}{4} = \frac{b}{2} = \frac{c}{2}$ => b=c 3a/4 = b/2 => 3a/2 = b

a <= 1/2
b <= 3/4
c <= 3/4

A: 1/2 * 20CPU, 1/2 * 18RAM = 10CPU, 9RAM
B: 12CPU, 6RAM
C: 9CPU, 9RAM

Pytanie 6

Częściowo poprawnie

Punkty: 1,00

Oflaguj pytanie

W bazie danych Hive utworzono tabelę **TAB** za pomocą poniższego polecenia:

```
CREATE TABLE TAB(
  B INT,
  C INT,
  D INT)
  PARTITIONED BY (A INT)
  CLUSTERED BY(B) INTO 25 BUCKETS
  STORED AS ORC;
```

Do tabeli TAB załadowano dane z pliku utworzonego za pomocą poniższego kodu

```
var tab_source = new ListBuffer[(Int,Int,Int,Int)]()
for (a <- (1 to 20); b <- (1 to 100); c <- (1 to 500); d <- (1 to 200))
   tab_source += ((a,b,c,d))
val tab_source_list = tab_source.toList
val tabRDD = sc.parallelize(tab_source_list)

tabRDD.toDF.write.csv("file:///C:\\tab_source.csv")
```

Załóż, że kolejność wierszy w tabeli **TAB** odpowiada implementacji generowania danych w pętli *for*.

Załóż, że serializowany rozmiar wartości w każdym wierszu każdej kolumny to **10B**.

Uzupełnij poniższe zadania:

- Liczba (**w milionach**) krotek (rekordów) w tabeli **TAB** wynosi: 200 ✔
- Rozmiar (**w milionach bajtów**) serializowanych źródłowych danych niepodzielonych w żaden sposób (na partycje lub kubełki) wynosi: 8000 ✔

Załóż, że tabela TAB w formacie ORC:

Załóż, że tabela TAB w formacie ORC:

- posiada funkcję rozpraszającą wiersze do poszczególnych kubełków jako ***klucz mod liczba_kubelkow***
- posiada stopień kompresji (wielkośćDanychPrzedKompresją/wielkośćDanychPoKompresji) serializowanych danych równy **2**.
- posiada wielkość paska wynoszącą (**w milionach bajtów**): **2**
- nie jest tabelą wspierającą ACID (nie obsługuje transakcji - poleceń DML) - nie zawiera dodatkowych katalogów wynikających z tej funkcjonalności

Uzupełnij poniższe zadania:

- Liczba katalogów wewnętrznych (poza głównym katalogiem) dla tabeli TAB wynosi: 20 ✔
- Liczba (**w milionach**) krotek wchodzących w skład pojedynczej partycji i pojedynczego kubełka wynosi: 0,4 ✔
- Wielkość (**w milionach bajtów**) plików wchodzących w skład pojedynczej partycji i pojedynczego kubełka wynosi (uwzględnij kompresję i wielkość serializowanych danych, pomiń wielkość indeksów): 6 ✔
- Liczba plików dla tabeli TAB wynosi: 500 ✔

Na tabeli **TAB** wykonano następujące zapytania:

Zapytanie A:

```
select *
from TAB
where A = 10
```

Uzupełnij poniższe zdania dotyczące efektów wykonania zapytania A:

- Liczba przeglądniętych plików: 25 ✔
- Liczba przeglądniętych statystyk (indeksów) na każdym z poziomów: 0 ✔

Zapytanie B:

Zapytanie B:

```
select *
from TAB
where B = 10
```

Uzupełnij poniższe zdania dotyczące efektów wykonania zapytania B:

- Liczba przeglądniętych plików: 25 ✖
- Liczba przeglądniętych statystyk (indeksów) na każdym z poziomów: 1050 ✖

Zapytanie C:

```
select *
from TAB
where C = 10
```

Uzupełnij poniższe zdania dotyczące efektów wykonania zapytania C:

- Liczba przeglądniętych plików: 250 ✖
- Liczba przeglądniętych statystyk (indeksów) na każdym z poziomów: 10500 ✖

Zapytanie D:

```
select *
from TAB
where A = 10 and B = 10 and C = 10
```

Uzupełnij poniższe zdania dotyczące efektów wykonania zapytania D:

- Liczba przeglądniętych plików: 1 ✔
- Liczba przeglądniętych statystyk (indeksów) na każdym z poziomów: 42 ✖

Pytanie 7

Poprawnie

Punkty: 1,00

Oflaguj pytanie

Które z poniższych nazw **nie jest** programem szeregowania zadań używanym przez system YARN

Wybierz jedną odpowiedź:

- a. LIFO ✔
- b. Capacity
- c. Fair
- d. FIFO

Pytanie 8

Częściowo poprawnie

Punkty: 1,00

Oflaguj pytanie

Programista Spark korzystając z klastra o następujących parametrach:

- liczba węzłów (wykonawców): **10**
- liczba dostępnych rdzeni procesorów na każdym z węzłów: **4**

utworzył zmienną RDD o nazwie `tabRDD` za pomocą poniższego polecenia:

```
var tab_source = new ListBuffer[(Int, Int, Int)]()
for (a <- (1 to 500); b <- (1 to 200); c <- (1 to 50))
    tab_source += ((a,b,c))
val tab_source_list = tab_source.toList


val tabRDD = sc.parallelize(tab_source_list)
```

BŁĄD ZADANIA

Załóż, że "kolejność" wierszy w utworzonej w ten sposób zmiennej jest nieokreślona.

Następnie programista wykonał następującą operację:

```
val aRDD1 = tabRDD.map(p => (p._3,p)).
                   filter(_._2._3 % 100 == 0)
```

```
                    filter(_._2._3 % 100 == 0)
val aRDD2 = aRDD1.groupByKey().
                    mapValues(ps => ps.map(gi => gi._2)).
                    mapValues(ps => ps.sum/ps.size)
aRDD2.collect()
```

Uzupełnij brakujące fragmenty w poniższym tekście.

- Liczba partycji w tabRDD wynosi 40 ✔
- Liczba rekordów w pojedynczej partycji tabRDD wynosi 125000 ✔
- Liczba partycji w aRDD1 wynosi 40 ✔
- Liczba rekordów w pojedynczej partycji aRDD1 wynosi 2500 ✖
- Liczba partycji w aRDD2 wynosi 40 ✖
- Liczba rekordów w pojedynczej partycji aRDD2 wynosi 1 ✔
- Liczba przesyłanych rekordów pomiędzy węzłami klastra: 2500 ✖

W związku ze słabą wydajnością powyższego kodu, programista dokonał jego rekonstrukcji i uruchomił poniższą operację:

```
val bRDD1 = tabRDD.map(p => (p._3, (p._3, 1))).
                    filter(_._2._1 % 100 == 0)
val bRDD2 = bRDD1.reduceByKey((p1,p2) => (p1._1 + p2._1, p1._2 + p2._2)).
                    mapValues(p => p._1/p._2)
bRDD2.collect()
```

Uzupełnij brakujące fragmenty w poniższym tekście.

- Liczba partycji w bRDD2 wynosi 40 ✖
- Liczba rekordów w pojedynczej partycji bRDD2 wynosi 1 ✔
- Liczba przesyłanych rekordów pomiędzy węzłami klastra: 10 ✖

9. Następnie powyższe programy zostały użyte w poniższym poleceniu uruchamiającym zadanie ***MapReduce***:

```
mapred streaming -D mapred.reduce.tasks=1 \
-files Identity.py, IntSum.py, IntSum.py \
-mapper Identity.py \
-combiner IntSum.py \
-reducer IntSum.py \
-input /user/maria_dev/input \
-output /user/maria_dev/output
```

W katalogu `/user/maria_dev/input` znajdowały się **dwa** identyczne pliki, których zawartość została utworzona w następujący sposób:

```
var tab_source = new ListBuffer[(String, Int)]()
for (a <- (1001 to 1010); b <- (1 to 10))
    tab_source += ((a.toString,b))
val tab_source_list = tab_source.toList
val tabRDD = sc.parallelize(tab_source_list)

tabRDD.toDF.repartition(1).write.csv("plik_wynikowy")
```

```
    tab_source += ((a.toString,b))
val tab_source_list = tab_source.toList
val tabRDD = sc.parallelize(tab_source_list)

tabRDD.toDF.repartition(1).write.csv("plik_wynikowy")
```

Ostatecznie przed przetwarzaniem format tych plików z csv został zmieniony na **tsv**.

Zwróć uwagę na następujące fakty:

- podczas uruchamiania nie określono typu klucza, a to oznacza, że przy wszelakim sortowaniu przez mechanizm MapReduce wykorzystany zostanie domyślny typ klucza - **Text** - np. liczby 1, 2, 10 zostaną posortowane: 1, 10, 2
- Hadoop Streaming dokonuje sortowania **tylko raz** - po mapowaniu. Agregator łączący (*combiner*) nie powinien zaburzać kolejności. Jeśli tak się stanie, kolejność danych przetwarzanych przez reduktor jest nieokreślona.
- sortowanie odbywa się tylko i wyłącznie w oparciu o jeden ze składników.

Uzupełnij brakujące wartości odnoszące się do zadania MapReduce uruchomionego za pomocą podanego powyżej polecenia:

- Liczba rekordów utworzonych w wyniku **mapowania**: 200 ✔
- Liczba rekordów **przesłanych** pomiędzy węzłami: 10 ✖
- Liczba rekordów umieszczonych w **pliku wynikowym**: 10 ✔
- **Kolejność** wierszy w pliku wynikowym jest:
  ☑ określona ✔ ☐ nieokreślona
- Jeśli kolejność wierszy jest określona podaj wartość **key** z pierwszej linii wynikowego pliku (w przeciwnym wypadku podaj wartość "x"): 1010 ✖
- Jeśli kolejność wierszy jest określona podaj wartość **value** z pierwszej linii wynikowego pliku (w przeciwnym wypadku podaj wartość "x"): 45 ✖