MapReduce

W ramach tutorialu spróbujemy dokonamy analizy danych *Cycle Share Dataset - Bicycle Trip Data from Seattle's Cycle Share System*:

- zawierających informacje dotyczące wykorzystania systemu rowerów miejskich w Seatle
- pochodzących z `https://www.kaggle.com/pronto/cycle-share-dataset`

Na początek będziemy chcieli odpowiedzieć na jedno proste pytanie:

Jaka jest wielkość (wyrażona za pomocą średniej liczby stanowisk) stacji rowerowych oddawanych do użytku w poszczególnych latach.

Zawartość danych to trzy zbiory danych zawierające informacje dotyczące: stacji rowerowych, przejazdów i pogody z lat 2014-2016. Wszystkie zbiory danych zawarte są w plikach CSV o następującej strukturze:

Stacje rowerowe (`station.csv`):

- `station_id`: identyfikator stacji
- `name`: nazwa stacji
- `lat`: szerokość geograficzna
- `long`: długość geograficzna
- `install_date`: data rozpoczęcia pracy stacji
- `install_dockcount`: liczba stanowisk rowerowych stacji w dniu instalacji
- `modification_date`: data modyfikacji stacji mającej wpływ na lokalizację lub liczbę stanowisk
- `current_dockcount`: liczba stanowisk rowerowych stacji w dniu 2016.08.31
- `decommission_date`: data wycofania stacji z eksploatacji

Przejazdy (`trips.csv`):

- `trip_id`: identyfikator przejazdu
- `starttime`: czas rozpoczęcia przejazdu w PST (Pacific Standard Time)
- `stoptime`: czas zakończenia przejazdu w PST
- `bikeid`: identyfikator roweru
- `tripduration`: czas trwania przejazdu
- `from_station_name`: nazwa stacji początkowej
- `to_station_name`: nazwa stacji końcowej
- `from_station_id`: identyfikator stacji początkowej
- `to_station_id`: identyfikator stacji końcowej
- `usertype`: typ użytkownika
  - `"Short-Term Pass Holder"` – użytkownik, który zakupił bilet krótkoterminowy (24-godzinny lub 3 dniowy);
  - `"Member"` – użytkownik, który zakupił bilet miesięczny lub roczny
- `gender`: płeć użytkownika
- `birthyear`: data urodzenia użytkownika

Danych dotyczących pogody nie będziemy tym razem wykorzystywali.

Przygotowanie środowiska

(5 minut)

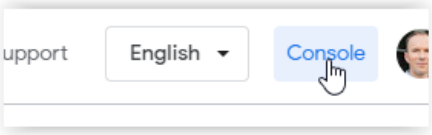
Na początku przygotujemy środowisko do realizacji tego zadania.

Nie będziemy instalowali własnego klastra Hadoop – skorzystamy ze środowiska Dataproc udostępnianego w ramach platformy Google Cloud Platform. Środowisko to jest prekonfigurowanym klastrem Hadoop z szeregiem dodatkowych opcji i narzędzi.

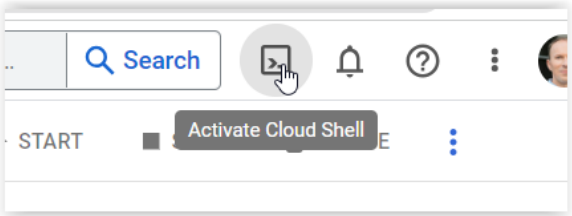
1. Utwórz klaster Dataproc korzystając z następujących kroków:

   a. Zaloguj się do wspomnianej platformy `https://cloud.google.com/gcp/`

   b. Przejdź do konsoli tej platformy



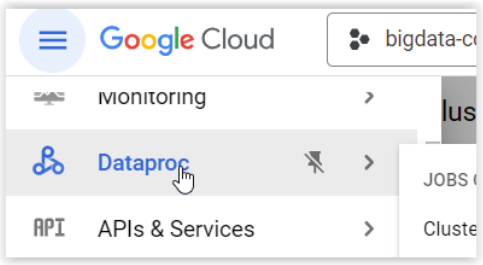
   c. Aktywuj *Cloud Shell* pozwalający zarządzać tę platformą za pomocą poleceń linii komend.

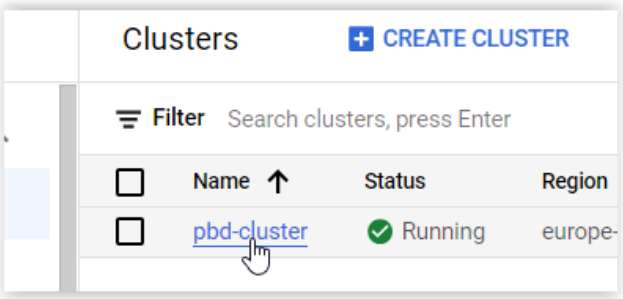


   d. Uruchom klaster Dataproc korzystając z poniższych poleceń *Cloud Shell*. Ustaw właściwe wartości zmiennych.

```
gcloud dataproc clusters create ${CLUSTER_NAME} \
 --enable-component-gateway --region ${REGION} \
 --master-machine-type n1-standard-4 --master-boot-disk-size 50 \
 --num-workers 2 --worker-machine-type n1-standard-2 --worker-boot-disk-size 50 \
 --image-version 2.1-debian11 \
 --project ${PROJECT_ID} --max-age=2h
```

   e. Wybierając z lewego menu pozycję *Dataproc* wyświetl listę klastrów. Zaczekaj aż klaster zostanie utworzony. Odśwież stronę i wybierz klaster utworzony przed chwilą



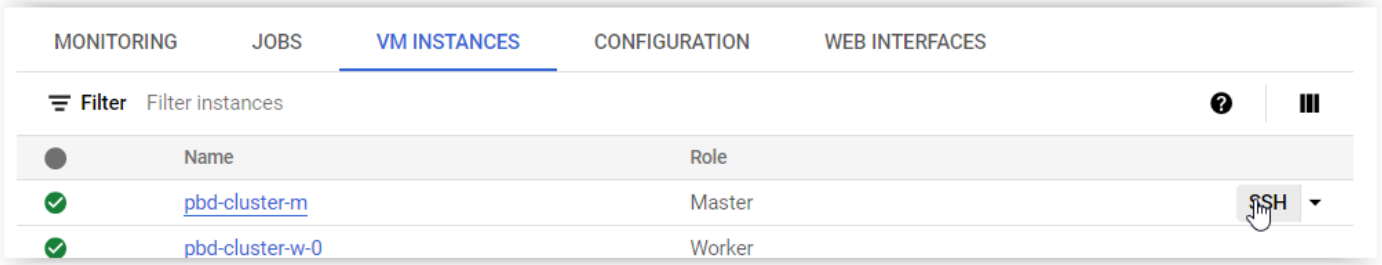


Przesłanie analizowanych danych

(15 minut)

Skoro mamy gotowe środowisko, w ramach którego będziemy analizować nasze dane, przyszedł czas na to, aby te dane tam umieścić. Każdorazowo nowy zbiór danych można umieścić w powiązanym z klastrem zasobniku, a następnie ładować go bezpośrednio z zasobnika.

2. Ze strony zawierającej szczegóły dotyczące klastra wybierz zakładkę *VM Instances*, a następnie podłącz się za pomocą terminala SSH do maszyny master klastra



3. Pobierz na lokalny dysk maszyny master `cycle-share-dataset.zip`

```
wget https://jankiewicz.pl/bigdata/hadoop-intro/cycle-share-dataset.zip
```

4. Rozpakuj plik `cycle-share-dataset.zip`, a następnie usuń pobrane archiwum

```
unzip cycle-share-dataset.zip

rm cycle-share-dataset.zip
```

```
jankiewicz_krzysztof@pbd-cluster-m:~$ unzip cycle-share-dataset.zip
Archive:  cycle-share-dataset.zip
  inflating: station.csv
   creating: trips/
  inflating: trips/trip1.csv
  inflating: trips/trip2.csv
  inflating: trips/trip3.csv
  inflating: weather.csv
jankiewicz_krzysztof@pbd-cluster-m:~$ rm cycle-share-dataset.zip
```

5. Załaduj potrzebne nam dane na zasobnik korzystając z poniższych instrukcji (jeśli nie masz utworzonego zasobnika w ramach Twojego kursu pomiń ten punkt).
   a. Za pomocą poniższego polecenia przypisz do zmiennej BUCKET_NAME nazwę Twojego zasobnika

```
export BUCKET_NAME={twoja_nazwa_zasobnika}
```

   b. Utwórz katalog `hadoop-intro/cycle-share-dataset` w ramach Twojego zasobnika

```
hadoop fs -mkdir -p gs://$BUCKET_NAME/hadoop-intro/cycle-share-dataset
```

   c. Następnie załaduj pobrane pliki do zasobnika powiązanego z naszym klastrem.

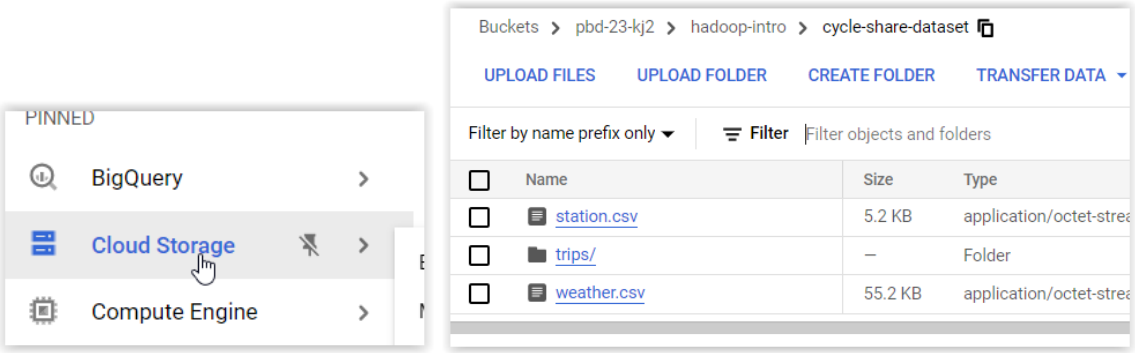
```
hadoop fs -put * gs://$BUCKET_NAME/hadoop-intro/cycle-share-dataset
```

d. Jeśli w przyszłości będzie potrzeba wykorzystania tych danych ponownie, można je będzie przekopiować bezpośrednio z zasobnika, lub nawet przetwarzać bezpośrednio z zasobnika. Sprawdź czy nasze dane znajdują się w docelowym miejscu

```
hadoop fs -ls gs://$BUCKET_NAME/hadoop-intro/cycle-share-dataset
```

```
Found 3 items
-rwx------   3 jankiewicz_krzysztof jankiewicz_krzysztof       5316 2023-10-01 14:03 gs://pbd-23-kj2/hadoop-intro/cycle-share-dataset/station.csv
drwx------   - jankiewicz_krzysztof jankiewicz_krzysztof          0 2023-10-01 14:03 gs://pbd-23-kj2/hadoop-intro/cycle-share-dataset/trips
-rwx------   3 jankiewicz_krzysztof jankiewicz_krzysztof      56516 2023-10-01 14:04 gs://pbd-23-kj2/hadoop-intro/cycle-share-dataset/weather.csv
```

e. Jeśli chcesz, możesz to także sprawdzić korzystając interfejsu sieciowego konsoli GCP.



6. Zanim przejdziemy dalej, zobaczmy jaki silnik wykonawczy wykorzystywany przez *MapReduce* w ramach wykorzystywanej konfiguracji. Generalnie silniki mogą być trzy: `local` – przydany dla lokalnego testowania zadań MapReduce, `yarn` – wiadomo i `classic` – wykorzystywany w Hadoop w wersji 1.X gdzie silnika YARN jeszcze nie było.
   Zobacz jaką wartość posiada własność `mapreduce.framework.name`.

```
hdfs getconf -confKey mapreduce.framework.name
```

```
jankiewicz_krzysztof@pbd-cluster-m:~$ hdfs getconf -confKey mapreduce.framework.name
yarn
```

7. Zobacz także jaką wartość posiada własność `mapreduce.job.reduces`. Za co odpowiada ten parametr? Parametr ten został "określony" przez inżynierów GCP na podstawie konfiguracji naszego klastra. Zastanów się, czy jego wartość została dobrana poprawnie. Jeśli masz wątpliwości przedyskutuj to z prowadzącycm.

```
hdfs getconf -confKey mapreduce.job.reduces
```

8. Podstawowy system plików wykorzystywany w ramach modelu MapReduce to HDFS, a nie lokalny system plików, w którym obecnie znajdują się nasze dane przeznaczone do analizy. Przenieśmy zatem dane do systemu HDFS. Utwórz w systemie plików HDFS katalog `input`.

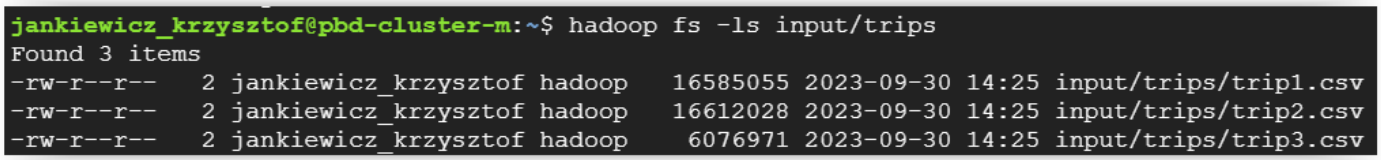
```
hadoop fs -mkdir -p input
```

9. Przekopiuj pliki tworzące nasz zestaw danych z lokalnego systemu plików do systemu plików HDFS do katalogu `input`

```
hadoop fs -put * input/
```

10. Sprawdź czy pliki znalazły się na swoim miejscu docelowym

```
hadoop fs -ls input/trips
```



11. Nasze pliki nie posiadają zbyt dużych rozmiarów. HDFS wykorzystuje z reguły bloki o rozmiarach 64-256MB. Nawet nasz największy plik nie posiada takiej wielkości. Zobaczmy na ile bloków został on podzielony.

```
hdfs fsck /user/$USER/input/trips/trip2.csv -files -blocks -locations
```

Analiza danych za pomocą języka Java

(30 minut)

WordCount

12. Zanim zaczniemy budowę naszego programu, spróbujemy uruchomić kanoniczny program MapReduce zliczający słowa (patrz np.: `https://www.cloudera.com/documentation/other/tutorial/CDH5/topics/ht_wordcount1.html`). Pobierz go za pomocą poniższego polecenia.

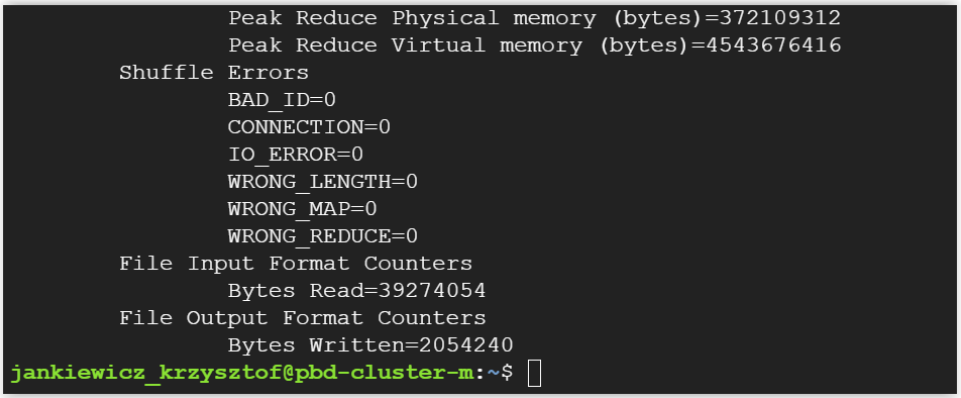
```
wget https://jankiewicz.pl/bigdata/hadoop-intro/WordCount.java
```

13. Przenalizuj jego zawartość. Dla wygody została ona podana poniżej

```
. . .
public class WordCount extends Configured implements Tool {

  private static final Logger LOG = Logger.getLogger(WordCount.class);

  public static void main(String[] args) throws Exception {
    int res = ToolRunner.run(new WordCount(), args);
    System.exit(res);
  }

  public int run(String[] args) throws Exception {
    Job job = Job.getInstance(getConf(), "wordcount");
    job.setJarByClass(this.getClass());
    // Use TextInputFormat, the default unless job.setInputFormatClass is used
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    return job.waitForCompletion(true) ? 0 : 1;
  }

  public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    private long numRecords = 0;
    private static final Pattern WORD_BOUNDARY = Pattern.compile("\\s*\\b\\s*");

    public void map(LongWritable offset, Text lineText, Context context)
        throws IOException, InterruptedException {
      String line = lineText.toString();
      Text currentWord = new Text();
      for (String word : WORD_BOUNDARY.split(line)) {
        if (word.isEmpty()) {
            continue;
        }
            currentWord = new Text(word);
            context.write(currentWord,one);
        }
    }
  }

  public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    public void reduce(Text word, Iterable<IntWritable> counts, Context context)
        throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable count : counts) {
        sum += count.get();
      }
      context.write(word, new IntWritable(sum));
    }
  }
}
```

14. Skompiluj oraz uruchom program korzystając z poniższych poleceń

```
mkdir -p build
javac -cp `hadoop classpath` WordCount.java -d build
jar -cvf wordcount.jar -C build/ .
hadoop jar wordcount.jar WordCount input/trips output
```



Wyniki znalazły się w nowoutworzonym katalogu `output` w systemie plików HDFS

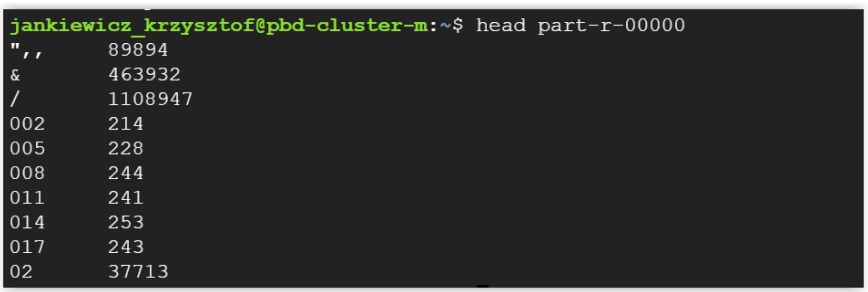
15. Zanim przeanalizujemy uzyskane wyniki odpowiedz na następujące pytania – skorzystaj ze statystyk wyświetlonych po wykonaniu zadania oraz analizy kodu programu i parametrów jego uruchomienia:
    a. Co było na wejściu do zadania *MapReduce* w naszym przypadku (jakie słowa on zliczał)?
    b. Jak wyglądają pary klucz-wartość generowane w wyniku użytej funkcji `map`?
    c. Ile rekordów wygenerowanych zostało na wyjściu funkcji `map`?
    d. Czy zastosowana została funkcja łącząca (agregator łączący - *combiner*) pomiędzy funkcją `map`, a `reduce` ograniczająca liczbę danych przesyłanych do funkcji `reduce`?
    e. Ile rekordów wygenerowała funkcja `reduce`?

16. Zobacz jak nazywają się pliki, które zostały umieszczone w wynikowym katalogu, a następnie pobierz jeden z nich do lokalnego systemu plików. Przy okazji skasuj katalog `output` w systemie HDFS.
    a. Ile jest plików wynikowych?
    b. Z czego wynika liczba tych plików?

```
hadoop fs -ls output
hadoop fs -copyToLocal output/part-r-00000 part-r-00000
hadoop fs -rm -r output
```

17. Zobacz uzyskane wyniki. Nie mają one nadzwyczajnego sensu, ale potwierdziły, że środowisko *MapReduce* jest gotowe do działania

```
head part-r-00000
```



Średnia wielkość stacji rowerowych

Czas na rozwiązanie naszego problemu. Dla przypomnienia, chcemy (w wyniku analizy pobranych przez nas danych) odpowiedzieć na jedno proste pytanie:

Jaka jest wielkość (wyrażona za pomocą średniej liczby stanowisk) stacji rowerowych oddawanych do użytku w poszczególnych latach.

Do odpowiedzi na to pytanie wystarczy przeanalizować zawartość pliku `station.csv`, który zawiera informacje o stacjach rowerowych. Plik ten ma następującą strukturę:

- `station_id`: identyfikator stacji
- `name`: nazwa stacji
- `lat`: szerokość geograficzna
- `long`: długość geograficzna
- `install_date`: data rozpoczęcia pracy stacji
- `install_dockcount`: liczba stanowisk rowerowych stacji w dniu instalacji
- `modification_date`: data modyfikacji stacji mającej wpływ na lokalizację lub liczbę stanowisk
- `current_dockcount`: liczba stanowisk rowerowych stacji w dniu 2016.08.31
- `decommission_date`: data wycofania stacji z eksploatacji

Zastanów się jakie dane wyjściowe będzie produkowała funkcja mapująca. Jakiego rodzaju operację będzie wykonywała funkcja redukująca.

Wydaje się, że dobrym rozwiązaniem byłoby, aby funkcja mapująca dla każdego napotkanego rekordu – linii w pliku źródłowym, tworzyła parę klucz-wartość o następującej zawartości: {rok z `install_date`}-{`install_dockcount`}. Następnie funkcja redukująca powinna wyliczyć średnią wartość {`install_dockcount`} dla każdego z kluczy.

Czy dałoby się użyć funkcji łączącej (agregatora łączącego)? Oczywiście, że tak. Jednak wymagałoby to zmiany formatu danych tworzonych przez funkcje mapującą. Wrócimy do tego problemu w swoim czasie.

18. Na początku zobacz jak wygląda zawartość linii w wysłanym pliku `station.csv`, a w szczególności format daty.

```
hadoop fs -head input/station.csv
```

```
jankiewicz_krzysztof@cluster-0999-m:~$ hadoop fs -head input/station.csv
"station_id","name","lat","long","install_date","install_dockcount","modification_date","current_dockcount","decommission_date"
"BT-01","3rd Ave & Broad St",47.618418,-122.350964,"10/13/2014",18,,18,
"BT-03","2nd Ave & Vine St",47.615829,-122.348564,"10/13/2014",16,,16,
"BT-04","6th Ave & Blanchard St",47.616094,-122.341102,"10/13/2014",16,,16,
"BT-05","2nd Ave & Blanchard St",47.61311,-122.344208,"10/13/2014",14,,14,
"CBD-03","7th Ave & Union St",47.610731,-122.332447,"10/13/2014",20,,20,
"CBD-04","Union St & 4th Ave",47.609221,-122.335596,"7/27/2015",18,,18,
```

Moglibyśmy nasz program napisać analogicznie do poprzedniego przypadku korzystając z edytora nano i poleceń z linii komend. Jednak tworzenie w ten sposób bardziej skomplikowanych programów może być po prostu trudne – analiza składni za pomocą programu vi nie dla wszystkich jest prosta. Dlatego tym razem nasz program napiszemy lokalnie, korzystając z *IntelliJ IDEA*.

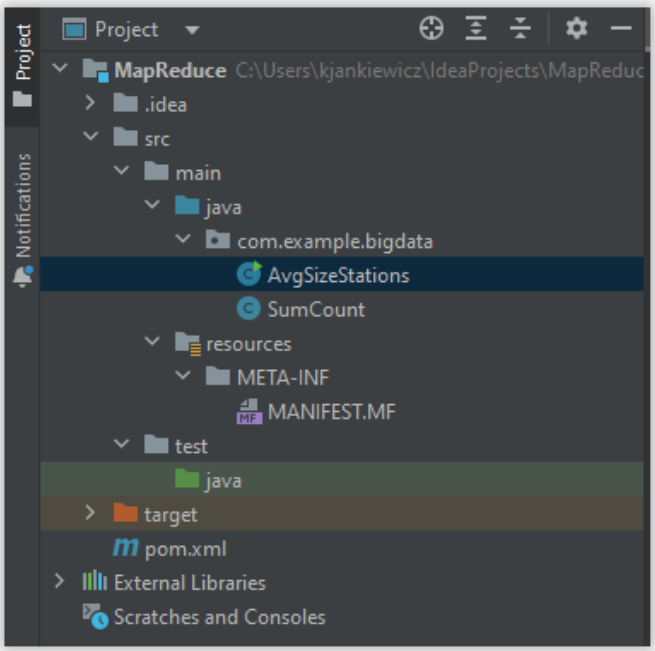
Konfiguracja projektu *IntelliJ IDEA*

19. Uruchom lokalnie *IntelliJ IDEA*

20. Pobierz na dysk lokalny i rozpakuj plik
    `http://jankiewicz.pl/bigdata/hadoop-intro/BD02_02-23-MapReduce.zip`

21. W *IntelliJ IDEA* otwórz rozpakowany katalog MapReduce jako projekt

22. Przenalizuj zawartość projektu



23. W terminalu za pomocą instrukcji `hadoop version` dowiedz się jaka wersja Hadoop wykorzystywana jest w docelowym środowisku

```
hadoop version
```

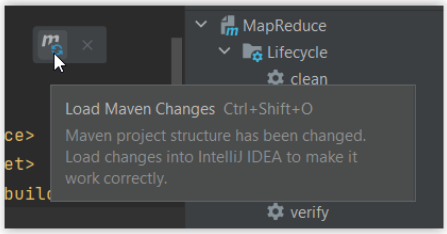
```
jankiewicz_krzysztof@pbd-cluster-m:~$ hadoop version
Hadoop 3.3.3
Source code repository https://bigdataoss-internal.googlesource.com/t
2ce9737b26578babcb1f4a
Compiled by bigtop on 2023-09-19T20:58Z
Compiled with protoc 3.7.1
From source with checksum 377b8aa9c43fbc94ba9470d7b240695
This command was run using /usr/lib/hadoop/hadoop-common-3.3.3.jar
```

24. Aby tworzyć aplikacje MapReduce potrzebujemy dwóch bibliotek:
    - *Apache Hadoop MapReduce Core*
    - *Apache Hadoop Common*

    Odnajdź właściwe wersje bibliotek na stronie `https://mvnrepository.com/`
    Zanotuj dla każdego z zestawów: `groupId`, `artifactId` oraz `version`.

25. Sprawdź czy zależności do tych bibliotek w pliku pom.xml zostały zdefiniowane prawidłowo, w razie potrzeby dokonaj stosownych poprawek i zaaplikuj zmiany.

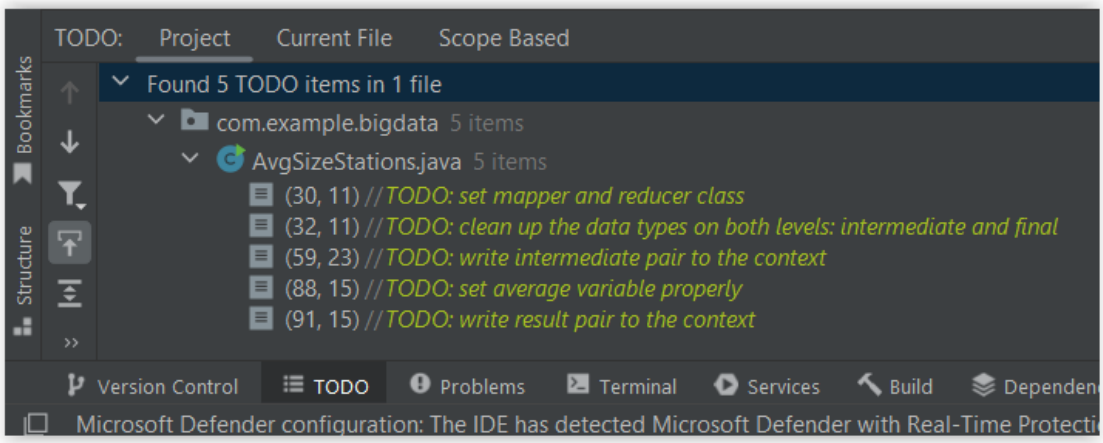
```
<properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceE
    <hadoop.version>3.3.3</hadoop.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>${hadoop.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-mapreduce-client-core</artifactId>
        <version>${hadoop.version}</version>
    </dependency>
```



Implementacja

26. Otwórz definicję klasy `AvgSizeStations`. Znajdź w nim:
    a. Mapper – definicję klasy wewnętrznej
    b. Reducer – definicję klasy wewnętrznej
    c. Definicja sterownika

27. Nasz program zawiera kilka brakujących fragmentów kodu, oraz jednego problemu, wymienionych na liście TODO

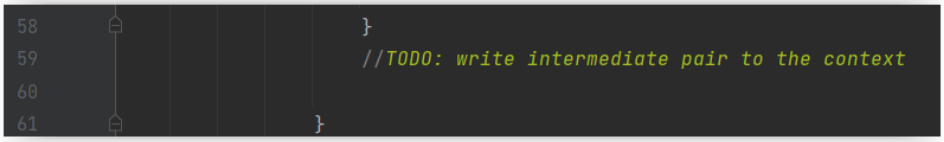


28. Zacznijmy od klasy Mapper. Znajdź jego definicję ponownie w naszym programie. W Javie implementuje interfejs Mapper i jest klasą generyczną z czterema parametrami formalnymi, które określają typy:
    - klucz wejściowy – w naszym przypadku: przesunięcie linii w pliku
    - wartość wejściowa – w naszym przypadku: wiersz tekstu
    - klucz pośredni – w naszym przypadku: powinien to być rok instalacji stacji – jest to tekst, ponieważ używamy go tylko jako klucza
    - wartość pośrednia – w naszym przypadku: jest to wielkość stacji – liczba.

    Przeanalizuj definicję tej klasy i odpowiedz na następujące pytania:

    - Jak nazywa się klasa, która implementuje funkcjonalność Mappera w naszym programie?
    - Jakiego typu (klasa) jest klucz wejściowy?
    - Jakiego typu (klasa) jest klucz pośredni?
    - Jakiego typu (klasa) jest wartość pośrednia?
    - Do której zmiennej tej klasy przypisana jest wartość pośrednia?
    - Który parametr metody map zawiera zawartość kolejnych wierszy z pliku źródłowego?
    - Jaki jest cel warunku: `if (offset.get() != 0)`

29. W treści tej klasy znajdziesz tylko jeden przedmiot TODO. Wprowadź poprawny brakujący fragment kodu poniżej.



30. Zamień TODO na DONE – usuwa ten element z listy TODO .

31. Przejdź do klasy *Reduktora*. Znajdź jego definicję ponownie w naszym programie. W Javie implementuje on interfejs `Reducer`, i jest to również klasa generyczna z czterema parametrami formalnymi, które określają typy:
    - klucz pośredni – klucz pośredni zwracany przez mapera lub przez... (wiesz co?)
    - wartość pośrednia – wartość pośrednia zwrócona przez mapera lub przez ... (wiesz co?)
    - klucz wyjściowy – w naszym przypadku będzie to tekst zawierający informację o roku
    - wartość wyjściowa funkcji redukcyjnej – w naszym przypadku średnia wielkość stacji – liczba.

    Przeanalizuj definicję tej klasy i odpowiedz na następujące pytania:

    a. Jak nazywa się klasa, która implementuje funkcjonalność reduktora w naszym programie?
    b. Jaki jest typ (klasa) klucza pośredniego?
    c. Jaki jest typ (klasa) klucza wyjściowego?
    d. Jaki jest typ (klasa) wartości wyjściowej?
    e. Jaką wartość zawiera parametr `key` metody `reduce` za każdym razem, gdy jest wywoływana?
    f. Jaką wartość zawiera parametr `values` metody `reduce` za każdym razem, gdy jest wywoływana?

32. W tej klasie znajdziesz dwie pozycje z listy TODO . Zacznijmy od pierwszego. Nasz program ma na celu obliczenie średniej liczby stanowisk na stacjach zainstalowanych w każdym kolejnym roku. Zmienna `average` powinna zawierać taką wartość dla każdego wywołania metody `reduce`. Wszystko, co jest potrzebne do ustawienia właściwej wartości tej zmiennej, jest już przygotowane. Wstaw odpowiedni brakujący fragment kodu, który przypisze poprawną wartość do zmiennej `average`.

```
87          }
88          //TODO: set average variable properly
89
90          resultValue.set(average);
```

Zamień TODO na DONE – usuwa ten element z listy TODO .

33. Drugi element TODO w tej klasie nie wymaga żadnego komentarza. Użyj kontekstu i jego metody `write`, aby wysłać poprawną parę na wyjście fazy redukcji.

```
90          resultValue.set(average);
91          //TODO: write result pair to the context
92
93      }
```

Zamień TODO na DONE – usuwa ten element z listy TODO .

34. Pozostały dwa ostatnie elementy TODO – znajdziesz je w miejscu konfiguracji zadania MapReduce.

    a. Jak nazywa się metoda, w której skonfigurowano zadanie MapReduce?
    b. Jak możemy określić dane wejściowe dla zadania MapReduce?
    c. Gdzie będą zapisywane dane wyjściowe MapReduce?
    d. Która klasa domyślnie będzie używana jako klasa Mapera?
    e. Jaka klasa domyślnie będzie używana jako klasa Reduktora?

35. Domyślne klasy mapera i reduktora nie będą zliczać średniego rozmiaru stacji dla każdego roku. Musimy korzystać z klas, które właśnie zaimplementowaliśmy. Uzupełnij ostatni brakujący fragment kodu naszego programu, ustawiając poprawne klasy mapera i reduktora.

```
29          FileOutputFormat.setOutputPath(job, new Path(args[1]));
30          //TODO: set mapper and reducer class
31
```

Zamień TODO na DONE – usuwa ten element z listy TODO .

36. Oprócz uzupełnionych brakujących fragmentów kodu, nasz program ma jedno poważne niedociągnięcie w sposobie konfigurowania zadania.

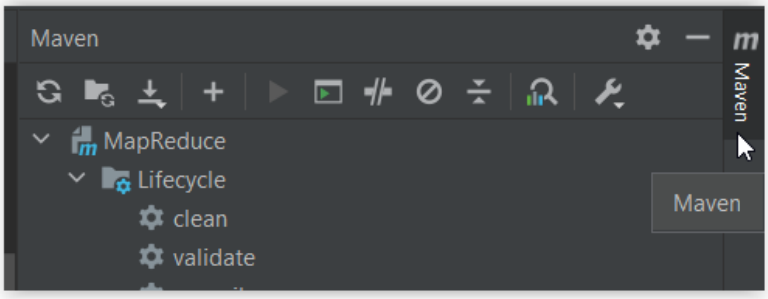
```
32          //TODO: clean up the data types on both levels: intermediate and final
33          job.setOutputKeyClass(Text.class);
34          job.setOutputValueClass(IntWritable.class);
```

37. Zapisz zmiany. Wszystkie błędy powinny zniknąć.

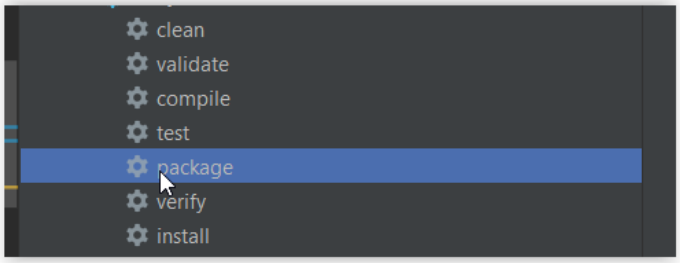
Wdrożenie na produkcji

Czas zbudować i skopiować nasz program do klastra Hadoop

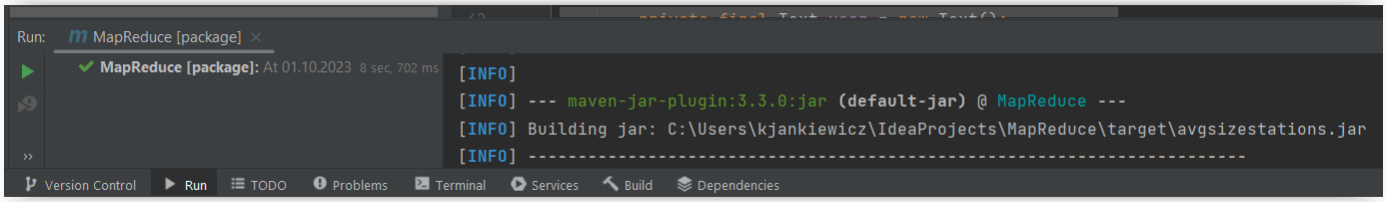
38. Otwórz panel narzędzia Maven



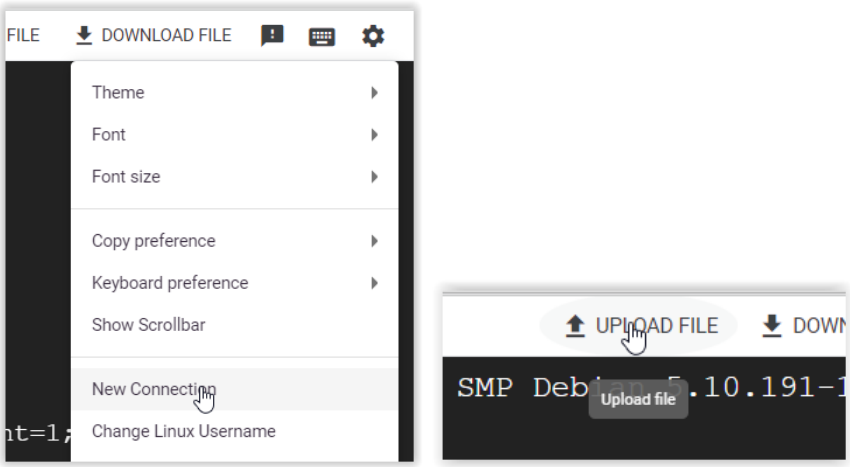
39. Wybierz dwukrotnie pozycje *package*, dzięki temu rozpocznie się budowa naszej aplikacji



40. Zlokalizuj miejsce w którym wygenerowany został plik jar



41. Możesz użyć funkcji terminala SSH (menu narzędzia), aby skopiować plik `avgsizestations.jar` do katalogu domowego lokalnego systemu plików na serwerze głównym naszego klastra. Dla bezpieczeństwa oryginalnego terminala zrób to w nowo nowym terminalu SSH.



42. W terminalu SSH sprawdź, czy plik `avgsizestations.jar` pojawił się w naszym domowym katalogu.

```
jankiewicz_krzysztof@pbd-cluster-m:~$ ls
WordCount.java      build          station.csv  weather.csv
avgsizestations.jar part-r-00000   trips        wordcount.jar
```

43. Uruchom nowy program *MapReduce*

```
hadoop jar avgsizestations.jar input/station.csv output
```

```
Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
File Input Format Counters
        Bytes Read=5316
File Output Format Counters
        Bytes Written=134
```

44. Pobierz dane wyjściowe zadania MapReduce do lokalnego systemu plików i przeanalizuj je.

```
hadoop fs -getmerge output stations.tsv
```

Możesz również wyświetlić dane zawarte w określonym pliku HDFS bezpośrednio za pomocą następującego polecenia. Pamiętaj, oba te polecenia mają sens tylko wówczas gdy wynik przetwarzania jest mały. To zdecydowanie nie jest standard w przypadku przetwarzania Big Data

```
hadoop fs -cat output/*
```

a. Co można powiedzieć o średniej wielkości stacji instalowanych każdego roku?
b. W którym roku średnia wielkość stacji różni się od wartości z pozostałych lat?
c. Ile lat obejmują dane w naszym pliku?

45. Zaglądnij do statystyk wyświetlanych po uruchomieniu zadania MapReduce.
    a. Ile stacji było w pliku wejściowym (nie licz nagłówka)?
    b. Ile par pośrednich zostało wysłanych między maperami a reduktorami?
    c. Ile jednostek zadań mapowania zostało uruchomionych?

Czas na dwa Twoje własne zadania.

Zadania własne

Wydajność – funkcja agregatora łączącego (Combiner)

46. W naszym projekcie MapReduce znajduje się klasa `SumCount`. Możesz jej użyć do obsługi wartości pośrednich, gdy nasze zadanie MapReduce używa agregatora łączącego. Co więcej, klasa agregatora łączącego jest również gotowa do użycia. Należy dostosować mapper, reduktor i sterownik, aby z niej skorzystać.
    a. Popraw nasz projekt i skorzystaj z agregatora łączącego.
    b. Wygeneruj, a następnie prześlij nową wersję pliku jar
    c. Usuń katalog wyjściowy utworzony przez poprzednie zadanie MapReduce

```
hadoop fs -rm -r output
```

    d. Uruchom nową wersję programu

```
hadoop jar avgsizestations.jar input/station.csv output
```

Wyniki powinny być identyczne. Na wszelki wypadek sprawdź zawartość wygenerowanych plików.

47. Zobacz do statystyk wyświetlanych po uruchomieniu zadania MapReduce.
    a. Ile par pośrednich zostało tym razem wysłanych między maperem (combinerami) a reduktorami?
    b. Czy ta liczba mogłaby się zmienić, gdyby uruchomiono więcej maperów (np. z powodu większej liczby plików wejściowych)?

Implementację zadań MapReduce za pomocą Javy nazywamy MapReduce Classic. Innym sposobem implementacji zadań MapReduce jest Hadoop Streaming. Dasz radę?

Hadoop Streaming

48. Zaimplementuj zadanie MapReduce za pomocą tej techniki. Użyj języka Python. Poniżej znajdziesz kilka przydatnych fragmentów kodu i poleceń.

```
wget https://jankiewicz.pl/bigdata/hadoop-intro/cos1.py

wget https://jankiewicz.pl/bigdata/hadoop-intro/cos2.py

mv cos*.py mapper.py

mv cos*.py reducer.py

python --version

chmod +x *.py

cat station.csv | python mapper.py | sort -k1,1 | python reducer.py

hadoop fs -rm -r output

tr -d "\r" < oldfile.py > newfile.py

mapred streaming \
 -files mapper.py,reducer.py \
 -input input/station.csv -output output \
 -mapper mapper.py -reducer reducer.py
```

49. Czy zadanie MapReduce zostało pomyślnie uruchomione przy użyciu usługi Hadoop Streaming? Jeśli tak, przyjmij gratulacje.
    a. Spójrz na wyniki. Czy są poprawne?
    b. Brakuje jednego polecenia w jednym z programów. Jakiego? W którym programie?

50. Popraw aktualną wersję programu i uruchom go ponownie.

51. (Dla ambitnych) Obecna wersja nie korzysta z agregatora łączącego. Postaraj się go wdrożyć.