

**WYDZIAŁ  
ELEKTROTECHNIKI  
I INFORMATYKI**  
POLITECHNIKI RZESZOWSKIEJ

Katedra Informatyki i Automatyki

**Jakub Iżewski**

Praktyczna weryfikacja możliwości  
wykorzystania modelu MapReduce do analizy  
danych

**Praca dyplomowa inżynierska**

Opiekun pracy:  
dr inż. Krzysztof Świder

**Rzeszów 2018**



## Spis treści

1.	Wstęp .....	5
2.	Wybrane elementy platformy Hadoop .....	7
2.1.	Schemat ogólny .....	7
2.2.	Rozproszony system plików HDFS .....	8
2.3.	System YARN oraz planowanie zasobów w klastrze .....	9
2.4.	Kolumnowa baza danych HBase .....	12
3.	Wprowadzenie do modelu MapReduce .....	15
3.1.	Definicja .....	15
3.2.	Działanie programów MapReduce w klastrze .....	17
3.3.	Implementacja programów na platformie Hadoop .....	20
3.4.	Instalacja oraz weryfikacja środowiska uruchomieniowego .....	23
4.	Przetwarzanie i analiza danych przy użyciu MapReduce .....	26
4.1.	Analiza plików tekstowych .....	26
4.2.	Łączenie danych pochodzących z plików tekstowych oraz bazy NoSQL .....	31
4.3.	Zapis danych do bazy NoSQL .....	38
5.	Analiza danych ankietowych .....	44
5.1.	Opis założeń oraz przygotowanie kodu źródłowego .....	44
5.2.	Przedstawienie rezultatów programu .....	50
5.3.	Wykorzystanie MapReduce jako API wyższego poziomu .....	52
5.4.	Dyskusja .....	52
6.	Podsumowanie .....	56
	Literatura .....	57
	Zawartość płyty CD .....	58



# 1. Wstęp

W obecnych czasach, na potrzeby aplikacji komercyjnych, portali społecznościowych oraz badań naukowych generowane, przechowywane oraz przetwarzane są petabajty danych. Coraz częściej wielkość danych, którymi należy zarządzać przerasta możliwości tradycyjnych baz relacyjnych pod kątem ich przechowywania oraz analizy. Istnieje zatem potrzeba użycia nowych technologii „radzących” sobie z ogromnymi ilościami, często nieustrukturyzowanych, danych. Chodzi przede wszystkim o: pozyskiwanie danych z różnych źródeł, przechowywanie ustrukturyzowanych lub nieustrukturyzowanych danych, zapewnienie dostępu oraz ochronę przed awariami. Pozyskiwanie oraz przechowywanie danych to jednak nie wszystko, gdyż coraz więcej korporacji i instytucji chce wykorzystywać dane do analizy w celu usprawniania procesów wspomagających podejmowanie decyzji biznesowych. Jednocześnie, coraz więcej instytucji naukowych chce przetwarzać duże zbiory danych do celów badawczych.

Celem pracy było wykonanie implementacji modelu MapReduce na platformie Hadoop oraz zbadanie jego możliwości w zakresie analizy danych. Ze względu na to, że sam model ściśle współpracuje z różnymi elementami ekosystemu Hadoop, do celów pracy należało również zbadanie innych narzędzi wchodzących w skład tego ekosystemu. W pracy skupiono się głównie na pokazaniu technicznych możliwości modelu, dlatego do części przykładów użyto sztucznie wygenerowanych danych.

Drugi rozdział pracy zawiera opis najważniejszych elementów platformy, które ściśle współpracują z modelem MapReduce. Przedstawiona została wysokopoziomowa architektura platformy oraz opisane jej wybrane elementy: system YARN służący do zarządzania zasobami klastra, rozproszony system plików HDFS oraz baza HBase. Każde z opisanych narzędzi jest odrębnym zestawem bibliotek typu open-source dostępnych w ramach platformy.

W rozdziale trzecim zaprezentowane zostały wybrane zagadnienia związane z modelem MapReduce oraz jego implementacją. Przedstawiono definicję samego modelu, założenia dotyczące kroków *Map* oraz *Reduce*, a także opcjonalne cechy modelu, których implementacja zależy głównie od użytkownika lub narzędzi wyższego poziomu. Kluczowym elementem przedstawionym w tym rozdziale jest sposób działania aplikacji MapReduce w klastrze komputerowym. Ze względu na fakt, że MapReduce jest napisany w języku Java, wszystkie przykłady w rozdziale trzecim jak i w dalszej części pracy są zaimplementowane w tym języku.

Czwarty rozdział pracy to przedstawienie technicznych możliwości MapReduce. Przygotowane zostały trzy programy przedstawiające różne aspekty modelu: sposoby odczytywania i zapisywania różnych formatów danych, implementację kroków *Map* oraz *Reduce*, implementację kroków opcjonalnych oraz komunikację modelu z bazą HBase.

W piątym rozdziale znajduje się przykładowa analiza danych ankietowych dotyczących dostępu społeczeństwa amerykańskiego do Internetu. Zaprezentowana została koncepcja wykorzystania modelu MapReduce jako API niższego poziomu dla innych narzędzi. Na podstawie otrzymanych rezultatów zostały przedstawione najważniejsze aspekty, które należy rozważyć przy implementacji MapReduce do rozwiązywania zadań związanych z analizą oraz przetwarzaniem danych.

Rozdział szósty zawiera krótkie podsumowanie pracy.

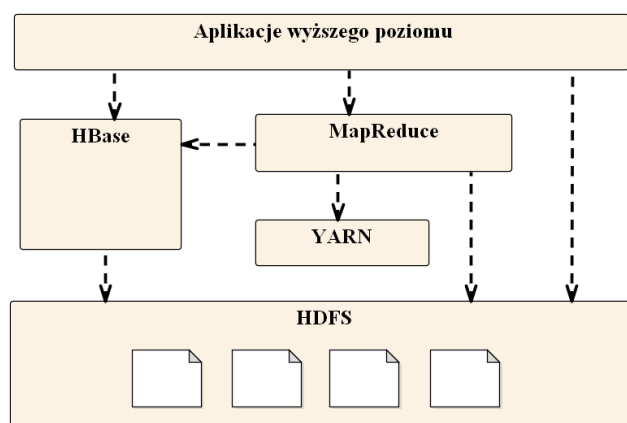
## 2. Wybrane elementy platformy Hadoop

### 2.1. Schemat ogólny

Hadoop to zbiór narzędzi open-source napisanych w języku Java, który umożliwia przechowywanie oraz przetwarzanie dużych zbiorów danych w sposób rozproszony. Do głównych komponentów platformy Hadoop zalicza się:

1. *Hadoop Commons* - moduł zawierający wszystkie biblioteki i pakiety pomocnicze, klasy oraz typy zmiennych używane przez pozostałe komponenty platformy
2. *Hadoop Distributed File System (HDFS)* - rozproszony system plików wzorowany na *Google File System*. System wykorzystuje wiele zaawansowanych mechanizmów aby zapewnić maksymalną skalowalność, niezawodność oraz odporność na błędy
3. *Hadoop YARN* - system do zarządzania zasobami w klastrze, odpowiedzialny za monitorowanie zasobów oraz przydzielanie ich do aplikacji
4. *Hadoop MapReduce* - implementacja modelu MapReduce, która umożliwia rozproszone wykonywanie zadań w klastrze.

Na przestrzeni ostatnich lat zostało wprowadzonych wiele innych komponentów platformy Hadoop, które umożliwiają wykonywanie różnych zadań w bardziej wydajny sposób, np. *Apache Spark*, *Apache Hive*, *Apache HBase*. Architektura wysokopoziomowa platformy została przedstawiona na rys. 2.1. Począwszy od wersji drugiej modelu, MapReduce wykorzystuje system YARN, który odpowiedzialny jest za kolejkovanie zadań oraz zarządzanie zasobami w klastrze.



Rys. 2.1. MapReduce w architekturze platformy Hadoop.

Programy MapReduce w większości przypadków korzystają z HDFS do pobierania danych, przechowywania dzienników zadań oraz zapisu danych wyjściowych. Mogą również wykorzystywać lokalne systemy plików lub rozwiązania chmurowe, jednakże nie są one zazwyczaj traktowane jako elementy platformy. Przykładem narzędzia, z którym MapReduce może bezpośrednio współpracować jest baza danych HBase. Biblioteki bazy HBase zawierają komponenty współpracujące bezpośrednio z programami MapReduce, a przykładem jest specjalnie przygotowany zestaw klas odpowiedzialnych za wydajny (wsadowy) zapis danych do tabel. Ponadto, model MapReduce udostępnia API, który może być wykorzystany przez narzędzia wyższego poziomu [1].

## 2.2. Rozproszony system plików HDFS

HDFS jest to rozproszony, abstrakcyjny system plików do przechowywania dużych zbiorów danych znajdujących się w klastrze komputerowym. System ten posiada architekturę master/slave, składa się z pojedynczego węzła nazw *NameNode* oraz dowolnej liczby węzłów danych *DataNodes*. Węzeł nazw (master) odpowiada za utrzymywanie przestrzeni nazw oraz regulowanie dostępu użytkowników do danych. Węzły danych (slaves) odpowiadają za fizyczne przechowywanie danych w klastrze [2].

Operacje na plikach odbywają się przy pomocy strumieni. HDFS opiera się na założeniu, że dane zostaną zapisane dokładnie jeden raz, a następnie będą służyły głównie do odczytu. Z racji tego, że do analizy potrzeba z reguły dużej części danych z danego pliku, pobieranie danych z HDFS wiąże się z opóźnieniem odczytu pierwszego rekordu na korzyść szybszego przetwarzania danych ze strumienia. Wynika to bezpośrednio z faktu, że do odczytania pojedynczego rekordu z pliku znajdującego się w HDFS potrzebne jest pobranie całego pliku. Bloki pamięci w systemie HDFS są dużo większe niż w przypadku zwykłych systemów operacyjnych, domyślną wartością dla HDFS jest 128 megabajtów. Aby uzyskać dostęp do danego fragmentu danych, należy przesłać odpowiednie zapytanie do węzła nazw. Posiadanie tak dużych bloków danych powoduje, że zmniejsza się liczba zapytań do węzła nazw oraz zwiększa się szybkość przetwarzania samych danych.

Użycie abstrakcyjnych bloków danych oraz zarządzającego nimi węzła nazw powoduje określone korzyści. Pliki mogą być większe niż dyski znajdujące się w klastrze, jeden blok danych może znajdować się na kilku dyskach, natomiast za jego spójność odpowiada węzeł nazw. Bloki danych dobrze działają z mechanizmami replikacji, co zwiększa odporność na awarie. W przypadku kiedy fragment pliku jest niedostępny na jednej z maszyn, dane mogą zostać pobrane z innej maszyny.

W tabeli 2.1. przedstawione zostały główne komendy platformy Hadoop służące do pracy z plikami w systemie HDFS. Większość z tych komend jest odwzorowaniem komend dostępnych w systemach operacyjnych np. możliwość przeglądania zawartości plików lub też tworzenia katalogów.

Komenda	Rezultat
<code>hadoop fs -ls files</code>	Wyświetla wszystkie pliki oraz podkatalogi w katalogu files w systemie plików HDFS, folder nadrzędny określony jest za pomocą parametru <code>fs.defaultFS</code> w plikach konfiguracyjnych Hadoop, np. <code>hdfs://localhost/</code> . Jest to identyfikator URI, który określa typ systemu plików (tutaj HDFS) oraz adres hosta i portu węzła nazw (tutaj po prostu localhost).
<code>hadoop fs -copyFromLocal file1.txt files/file2.txt</code>	Kopiuje plik z obecnego folderu do HDFS replikując przy tym dany plik tyle razy ile jest to określone w parametrze <code>dfs.replication</code> w jednym z plików konfiguracyjnych Hadoopa. Wartość ta nie może być mniejsza niż 1.
<code>hadoop fs -copyToLocal files/file2.txt file3.txt</code>	Kopiuje plik z systemu HDFS do lokalnego folderu. Warto zaznaczyć, że HDFS wyszukuje dany plik na podstawie metadanych po blokach. Plik jest odtwarzany z bloków znajdujących się w systemie.
<code>hadoop fs -appendToFile file1.txt files/file4.txt</code>	Dodaje pojedynczy plik źródłowy z systemu lokalnego do danego pliku w systemie HDFS.
<code>hadoop fs -mkdir katalog</code>	Tworzy nowy katalog.
<code>hadoop fs -move localFile HadoopFile</code>	Przenosi lokalny plik do systemu HDFS.
<code>hadoop fs -rm HadoopFile</code>	Usuwa plik z systemu HDFS oraz wszystkie jego repliki.

Tabela 2.1. Zestaw wybranych komend platformy Hadoop umożliwiające komunikację z systemem HDFS.

Kluczowym elementem są tutaj komendy służące do przekazywania plików z, lub do, lokalnego systemu plików *copyFromLocal* oraz *copyToLocal*. Pozwalają one na współpracę z wieloma systemami plików np. HDFS lub też *Azure Blob Storage*, który jest jednym z rozwiązań chmurowych firmy Microsoft. W przypadku kopiowania plików do systemu HDFS są one replikowane i zapisywane na różnych serwerach. Zwiększa to dostępność plików oraz odporność na błędy.

### 2.3. System YARN oraz planowanie zasobów w klastrze

YARN *Yet Another Resource Negotiator* to używany na platformie Hadoop system, który zarządza zasobami klastra. Dodatkowo, udostępnia API do zgłaszania żądań, monitorowania zasobów oraz aplikacji. YARN z reguły nie jest wywoływany bezpośrednio przez użytkowników Hadoopa, użytkownicy z reguły korzystają z API udostępnianych przez

programy wyższego poziomu zaimplementowane na platformie Hadoop, jak np. MapReduce lub Spark. Początkowo system był dedykowany do przydzielania zasobów dla procesów MapReduce, jednakże okazał się on na tyle uniwersalny, że bardzo wiele nowych komponentów używa tego systemu do uruchamiania procesów lub zadań w klastrze [3].

Kluczowym elementem YARN jest menadżer zasobów *Resource Manager*, który odpowiada za zarządzanie wszystkimi zasobami klastra oraz wspiera wszystkie aplikacje, które uruchamiane są w systemie YARN. Menadżer zasobów posiada informacje o wszystkich węzłach roboczych oraz o dostępnych zasobach, posiada wiele wbudowanych komponentów oraz serwisów do zarządzania zasobami klastra oraz bezpieczeństwem. Kluczowymi jego komponentami są:

1. Planista zasobów *Resource Scheduler* - główny komponent menadżera zasobów, planista jest odpowiedzialny za sposób i ilość zasobów przydzielonych do konkretnego procesu
2. Komponenty służące do monitorowania procesów oraz wymiany informacji pomiędzy menadżerem zasobów a węzłami, np. *Resource Tracker Service*
3. Komponenty służące do wymiany informacji oraz żądań z menadżerem aplikacji *Application Manager*
4. Komponenty odpowiedzialne za uwierzytelnianie żądań.

W systemie YARN można skonfigurować kilka rodzajów planisty zasobów. Pierwszym rodzajem jest planista, który przydziela zasoby do zadań zgodnie z kolejką FIFO, żądania o przydzielenie zasobów obsługiwane są w kolejności zgłoszenia oraz nie istnieją procesy o wyższym lub niższym priorytecie. Ten rodzaj planisty jest skuteczny w przypadku małych procesów, które mogą być szybko obsługiwane, jednakże brak odpowiedniego planowania oraz brak możliwości ustalania priorytetów zadań może powodować problemy wydajnościowe w przypadku większych programów lub procesów o wysokim znaczeniu biznesowym.

Kolejnym rodzajem planisty jest *Capacity Scheduler*. Umożliwia on bezpieczne przydzielanie zasobów poszczególnym grupom korzystającym z klastra. Jego konfiguracja opiera się na zdefiniowaniu jednej lub więcej kolejek razem z odpowiednią ilością zasobów (np. liczby procesorów). Konfiguracja kolejek odbywa się poprzez zdefiniowanie odpowiednich plików XML. Głównym plikiem konfiguracyjnym jest plik *capacity-scheduler.xml*, który odpowiada za definiowanie kolejek oraz podstawowych parametrów

każdej z nich. *Capacity Scheduler* posiada zdefiniowaną kolejkę nadrzędną *root queue* natomiast każda inna kolejka jest kolejką-dzieckiem *child queue* kolejki nadrzędnej [4].

Na listingu 2.1. znajduje się przykładowy plik XML definiujący jedną kolejkę *userqueue* oraz kilka podstawowych parametrów:

1. `yarn.scheduler.capacity.root.queues` – parametr informujący o tym, jakie kolejki dziedziczą po danej kolejce nadrzędnej, w tym przypadku jest to tylko kolejka *userqueue*
2. `yarn.scheduler.capacity.root.userqueue.capacity` - procentowa suma pojemności wszystkich kolejek na danym poziomie, w tym przypadku równa 100, jako, że nie ma innych kolejek dziedziczących po kolejce *root*
3. `yarn.scheduler.capacity.root.userqueue.user-limit-percent` – maksymalna procentowa wartość zasobów jaką mogą dzielić użytkownicy w danej kolejce. Wartość 20 oznacza, że aktywnych może być maksymalnie 5 użytkowników danej kolejki dzielących dokładnie 20% wszystkich zasobów. Kolejny użytkownik danej kolejki będzie musiał poczekać na zwolnienie zasobów.

```
<property>
  <name>yarn.scheduler.capacity.root.queues</name>
  <value>userqueue</value>
  <description>
    The queues at the this level (root is the root queue).
  </description>
</property>
<property>
  <name>yarn.scheduler.capacity.root.userqueue.capacity</name>
  <value>100</value>
  <description>Default queue target capacity.</description>
</property>
<property>
  <name>yarn.scheduler.capacity.root.userqueue.user-limit-percent</name>
  <value>20</value>
</property>
```

Listing 2.1. Przykładowy plik definiujący kolejki planisty zasobów.

Kolejnym rodzajem planisty jest *fair scheduler*. Działanie tego planisty polega na przydzieleniu równej liczby zasobów każdej z aplikacji. Kiedy w klastrze uruchomiona jest jedna aplikacja to może ona zażądać zasobów całego klastra. Kiedy uruchamiana jest kolejna aplikacja, podział zasobów pomiędzy nimi powinien być jednakowy. W tym modelu również można konfigurować informacje takie jak np. wagi kolejek (kolejka z większą wagą

otrzymuje więcej zasobów), jednakże idea jest taka, aby wszystkie zasoby rozdzielać tak równomiernie między aplikacjami jak tylko się da.

Kolejnym elementem systemu YARN są tzw. kontenery. Każdy kontener jest kolekcją fizycznych zasobów wewnątrz danego węzła, np. liczba pamięci RAM albo liczba wirtualnych rdzeni procesora. Każdy węzeł może posiadać jeden lub wiele kontenerów. Kontenery są utrzymywane przez menadżera węzła i planowane do wykorzystania przez planistę zasobów. Każda aplikacja wykorzystuje pewną liczbę kontenerów, przy czym za kontener 0 uważa się kontener zarządcy aplikacji *Application Master Container*.

Menadżer węzła *NodeManager* jest agentem systemu YARN znajdującym się w każdym węźle. Do jego głównych zadań należą: utrzymywanie połączenia z planistą zasobów, dogłębne oglądanie kontenerów znajdujących się wewnątrz danego węzła, monitorowanie zasobów, utrzymywanie dzienników węzła oraz komunikację z aplikacjami pomocniczymi.

Zarządca aplikacji *Application Master* jest procesem, który koordynuje wykonanie danego procesu lub aplikacji w klastrze. Każda aplikacja posiada własnego zarządcę, który negocjuje zasoby i kontenery z planistą zasobów. Zarządca współpracuje również z menadżerami węzłów w celu wykonania oraz monitorowania poszczególnych zadań [5].

## 2.4. Kolumnowa baza danych HBase

HBase jest to model bazy danych na platformie Hadoop za pomocą którego można uzyskać w szybki sposób dostęp do konkretnych danych w bardzo dużych ustrukturyzowanych zbiorach. Przyczyna powstania HBase wiąże się z faktem, że pobieranie danych z HDFS może odbywać się jedynie w sposób sekwencyjny tzn. aby uzyskać dostęp do małej porcji danych należy przeskanować cały konkretny zbiór danych.

HBase jest bazą danych NoSQL opartą o model rodziny kolumn. Pojęcie NoSQL odnosi się do baz, w których dane przechowywane oraz wyszukiwane są w inny sposób niż w przypadku baz relacyjnych. Bazy NoSQL nie muszą spełniać reguł ACID. Wynika to bezpośrednio z faktu, że dane są replikowane pomiędzy węzłami. W przypadku awarii dane mogą być niespójne pomiędzy węzłami, natomiast system powinien dążyć do ich synchronizacji. Cechy te znane są jako reguły BASE. Skrót BASE pochodzi od słów: *Basically available, Eventual consistency* oraz *Soft state*.

W każdej tabeli bazy HBase mamy zdefiniowane jedną lub więcej rodzin kolumn *column family*, które zawierają dowolną liczbę kolumn. Każda kolumna posiada dane w postaci klucz-wartość. Dane w konkretnych rodzinach kolumn są zapisane w sposób ciągły





na dysku. Dodatkowo, rekordy również można pogrupować w tzw. regiony, są to po prostu pewne przedziały wartości kluczy, np. jednym regionem mogą być nazwiska osób, których pierwsza litera mieści się w przedziale od A do M, natomiast drugim regionem mogą być nazwiska osób z przedziału od N do Z [6].

Przykładowa organizacja struktury tabeli w bazie HBase została przedstawiona na rys. 2.2. Tabela posiada dwie rodziny kolumn, z których pierwsza ma cztery kolumny oraz druga posiada 3 kolumny, dodatkowo tabela posiada dwa regiony, które zawierają rekordy pochodzące z odpowiedniego zakresu identyfikatora. W odróżnieniu od tabel relacyjnych, gdzie dany rekord musi posiadać wartość dla każdej z kolumn w tabeli, lub wartość null w przypadku jej braku, w bazie HBase może podać definicję tylko dla wybranych kolumn, np. rekord z identyfikatorem 1 może nie określać żadnej wartości dla kolumn 1 i 3.

	Identyfikator	Rodzina kolumn 1				Rodzina kolumn 2		
		Kolumna1	Kolumna2	Kolumna3	Kolumna4	Kolumna5	Kolumna6	Kolumna7
Region1	1							
	2							
	3							
	4							
Region2	5							
	6							
	7							
	8							
	9							

Rys. 2.2. Przykładowa struktura tabeli HBase.

Dzięki wprowadzeniu rodzin i regionów można w łatwy sposób umieścić dane na różnych serwerach. Dzięki takiemu zabiegowi skalowalność tabeli (systemu) jest bardzo prosta i ma charakter liniowy. Na rys 2.3. przedstawiono w jaki sposób można zorganizować strukturę tabeli HBase w klastrze komputerowym. Każdy serwer na którym znajduje się przynajmniej jeden region nazywany serwerem regionu *region server*. Na jednym serwerze regionu może znajdować się dowolna liczba regionów, jak i jeden region może obejmować wiele serwerów.

	Identyfikator	Rodzina kolumn 1				Rodzina kolumn 2		
		Kolumna1	Kolumna2	Kolumna3	Kolumna4	Kolumna5	Kolumna6	Kolumna7
Region1	1							
	2							
	3							
	4							
Region2	5							
	6							
	7							
	8							
	9							

Rys. 2.3. Przykładowa implementacja tabeli bazy HBase w klastrze.

Dane z tabeli HBase mogą być trzymane zarówno w lokalnym systemie plików (w przypadku samodzielnej bazy HBase) lub systemie plików HDFS. Dzięki takiej alokacji danych baza HBase bardzo dobrze współpracuje z modelem MapReduce, gdzie jednocześnie możemy uzyskać niewielkie zestawy danych z wielu różnych maszyn, np. przydzielając jedno zadanie mapowania na konkretny region i rodziny kolumn. Operacje na bazie HBase można wykonywać komunikując się z odpowiednio przygotowanym API np. przy użyciu języka Java. HBase udostępnia również interaktywną powłokę (podobną np. do terminali linuxowych). W tabeli 2.2. znajduje się lista kluczowych komend do pracy z bazą HBase.

Komenda	Opis
Status	Udostępnia podstawowe informacje o bazie HBase, np. liczba serwerów.
create 'table_name','column_family'	Tworzy tabelę o podanej nazwie oraz o zadanych rodzinach kolumn.
List	Tworzy listę wszystkich tabel dostępnych w bazie HBase.
alter	Modyfikuje strukturę tabeli, np. zwiększa maksymalną liczbę krotek dostępnych w danej rodzinie kolumn, usuwa rodzinę kolumn z danej tabeli.
drop	Usuwa tabelę z HBase wraz z danymi, przed usunięciem tabeli należy ją najpierw wyłączyć, dostępna jest również komenda drop_all, usuwa ona wszystkie tabele, których nazwa pasuje do wyrażenia regularnego podanego jako parametr po komendzie.
put	Dodaje rekord do konkretnej kolumny w konkretnej tabeli, aby dodać rekord należy podać informacje takie jak klucz danego wiersza, rodzinę kolumn oraz tabelę.
Get	Pobiera informacje z danej tabeli dla danego klucza, możliwe jest pobranie całego rekordu lub tylko wartości z jednej z kolumn.
delete	Usuwa konkretną krotkę, możliwe jest również usunięcie całego wiersza poprzez podanie komendy deleteall.
scan	Pobiera dane z tabeli, poprzez dodanie odpowiedni parametrów np. LIMIT, FILTER, STARTROW, STOPROW możliwe jest wybranie tylko konkretnych rekordów z całej tabeli, w przypadku braku parametrów pobierane są dane z całej tabeli.
grant	Dodaje konkretne uprawnienia danemu użytkownikowi do danych, umożliwia przyznawanie uprawnień na poziomie tabel lub kolumn.
revoke	Usuwa konkretne uprawnienia danemu użytkownikowi, podobnie jak w przypadku grant, pozwala na przyznawanie uprawnień na poziomie tabel lub kolumn.

Tabela 2.2. Zestawienie wybranych komend bazy HBase.

### 3. Wprowadzenie do modelu MapReduce

#### 3.1. Definicja

MapReduce to model obliczeń matematycznych za pomocą którego w sposób równoległy można przetwarzać duże zbiory danych. Model polega na podziale przetwarzania na etapy mapowania i redukcji. Założenia modelu można wyrazić przy pomocy następującej formuły:

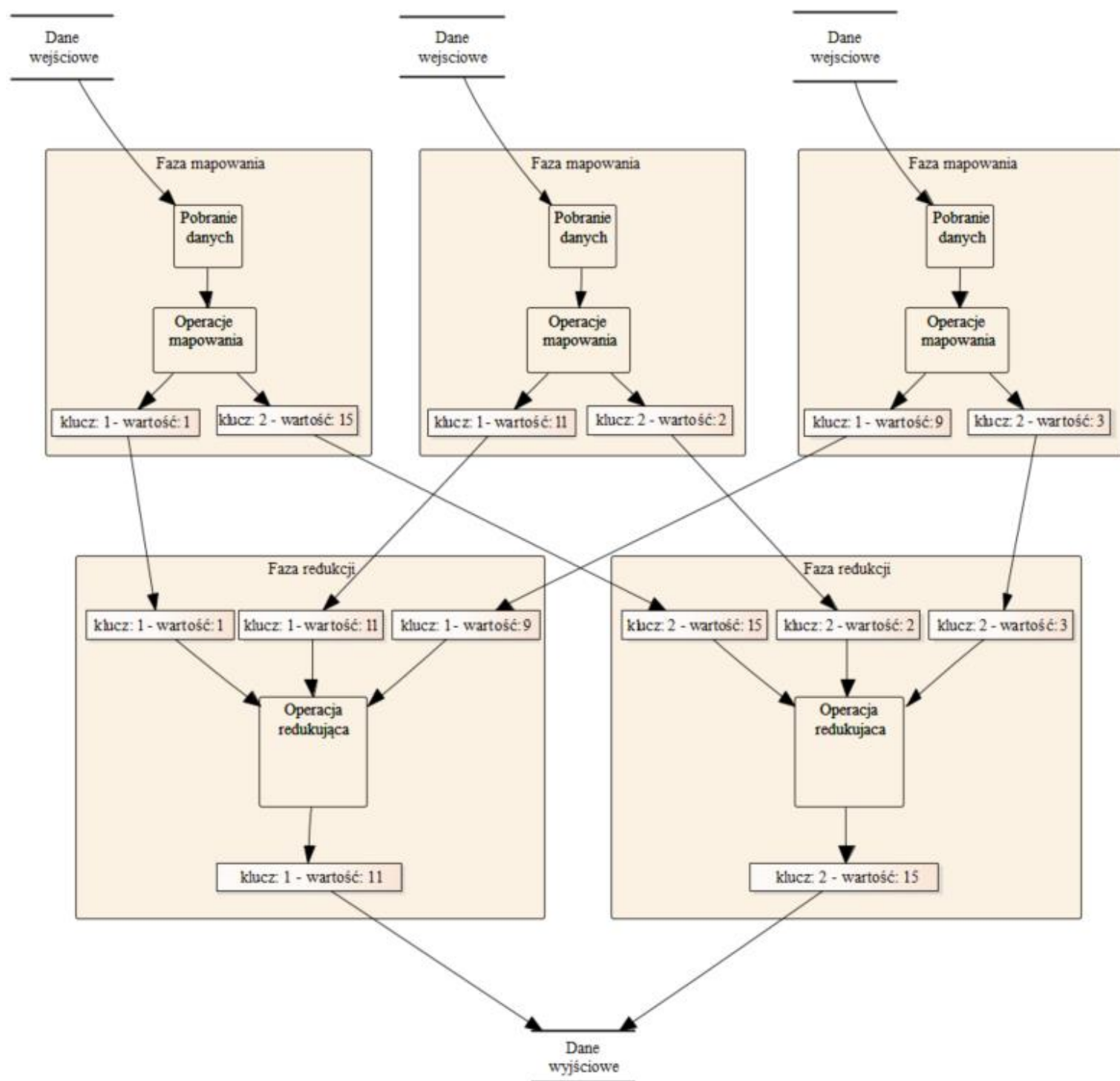
$$\begin{aligned} \text{Map}(k1, v1) &\rightarrow \text{list}(k2, v2) \\ \text{Reduce}(k2, \text{list}(v2)) &\rightarrow \text{list}(k3, v3) \end{aligned}$$

Na każdym etapie dane wejściowe i wyjściowe są parami kluczy i wartości. Każda operacja mapująca dostaje na wejściu pewien zbiór danych i wyodrębnia z niego dane postaci klucz-wartość, które przekazywane są dalej do operacji redukujących. Te z kolei dla każdej wartości klucza otrzymują listę wartości z poszczególnych operacji mapujących, dokonują redukcji i przedstawiają dane również w formie klucz-wartość. Podczas przetwarzania liczba zadań mapujących i redukujących może przybierać różne wartości i zależy głównie od rozmiaru zadania, dostępnego sprzętu oraz od parametrów i funkcji napisanych przez programistę.

Na rys. 3.1. przedstawiony został schemat realizacji przykładowego zadania polegającego na znalezieniu maksymalnej wartości dla danego klucza. Operacje mapowania przekształcają plik wejściowy na dane w postaci klucz-wartość, gdzie kluczem jest identyfikator oraz wartością jest liczba całkowita. W zależności od wartości klucza dane przesyłane są do odpowiednich reduktorów, które odpowiedzialne są za połączenie rezultatów oraz znalezienie maksymalnej wartości. Rezultaty wszystkich obliczeń są również w postaci klucz-wartość.

Z modelem związane są pewne formalne wymagania, które muszą zostać spełnione. Programy MapReduce grupują dane oraz przetwarzają ich fragmenty niezależnie więc kolejność ich przetwarzania nie może wpływać na rezultaty obliczeń. Dodatkowo, puste zbiory danych nie mogą wpływać na działanie modelu. Posiadając odpowiednie zbiory A, B i C, pusty zbiór E oraz sumę zbiorów oznaczoną jako \*, działanie modelu MapReduce musi spełniać następujące zależności matematyczne:

- i)  $A * (B * C) = (A * B) * C$
- ii)  $E * A = A * E = A$



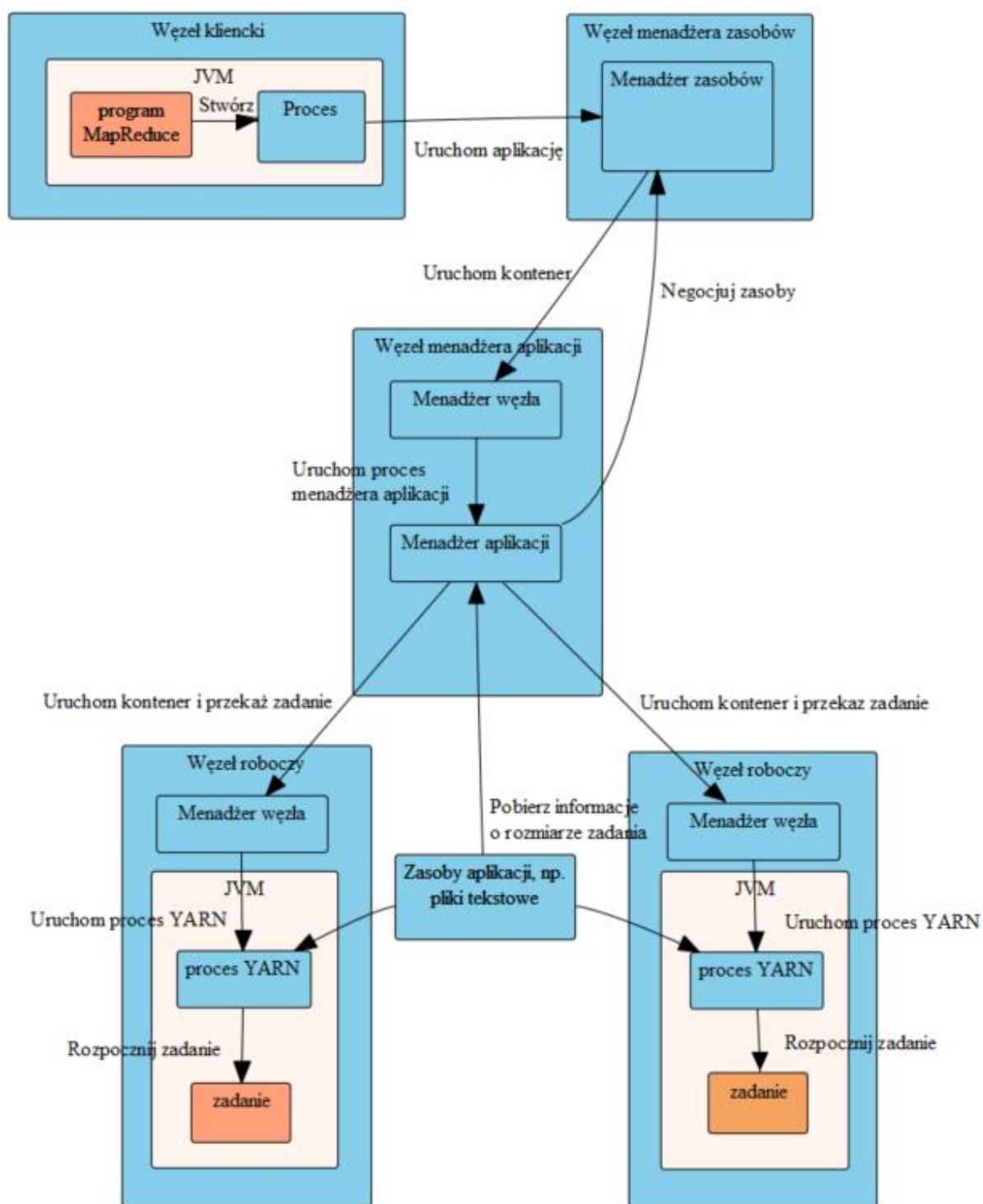
Rys. 3.1. Schemat przykładowego zadania MapReduce.

W zależności od implementacji można wyróżnić również inne fazy modelu. Faza partycjonowania *partitioning* przebiega pomiędzy operacjami mapowania i redukcji, dzieli dane wyjściowe operacji mapującej na partycje na podstawie kodu użytkownika. Do każdej partycji przydzielone jest jedno zadanie redukujące. Faza łączenia *combining* występuje kiedy na wyjściu każdego zadania mapującego aplikuje się dodatkową logikę redukującą, przed przystąpieniem do dalszej analizy. Operację tą wykonuje się aby ograniczyć liczbę danych przesyłanych pomiędzy węzłami. Kolejną fazą, która może występować w modelu jest przesunięcie i sortowanie *shuffle & sort*, faza ta może występować przed fazą redukcji, kiedy dane są sortowane oraz grupowane po wartościach klucza aby usprawnić działanie fazy redukującej. Ostatnią z dostępnych faz jest filtrowanie. Nie wszystkie wartości uzyskane podczas operacji mapowania muszą być wzięte do analizy więc mogą być odfiltrowane na każdym etapie procesu.

### **3.2. Działanie programów MapReduce w klastrze**

Działanie programu MapReduce na platformie Hadoop w modelu rozproszonym zostało przedstawione na rysunku 3.2. Cały proces rozpoczyna się od węzła klienckiego, może to być lokalna maszyna klienta lub też jedna z maszyn dostępnych w klastrze. Klient tworzy program MapReduce i wysyła żądanie uruchomienia programu. Przed wysłaniem żądania klient może przekazać zasoby potrzebne do wykonania zadania, np. część kodu bądź pliki z danymi. Żądanie wykonania programu MapReduce trafia do węzła menadżera zasobów, który ma informacje o zasobach dostępnych w klastrze. Warto dodać, że menadżer zasób, menadżer węzła oraz wiele innych serwisów udostępnia odpowiednie API, za pomocą których można pobrać informacje dotyczące aktywności w klastrze, np. liczba aktywnych aplikacji czy liczba aktywnych kontenerów [7].

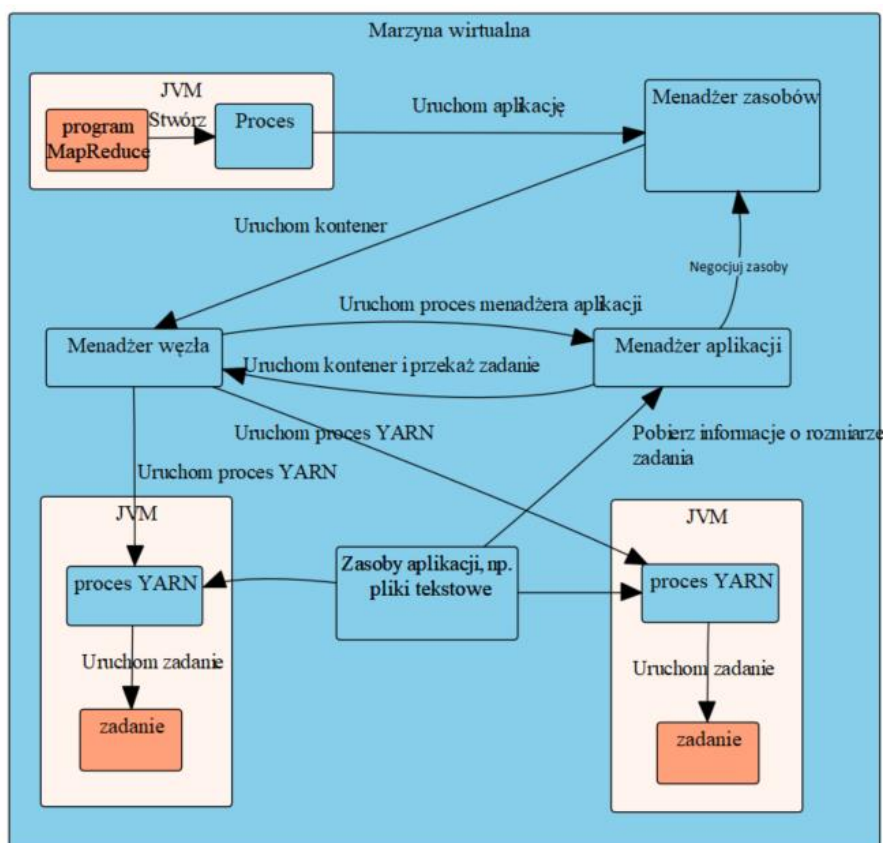
Menadżer zasobów po otrzymaniu zgłoszenia łączy się z jednym z dostępnych węzłów roboczych, który przydziela kontener na potrzeby menadżera aplikacji. Od tej chwili odpowiedzialność za poprawne wykonanie danego programu spoczywa na menadżerze aplikacji, który rozdziela zadania, monitoruje statusy poszczególnych zadań, a także negocjuje zasoby z menadżerem zasobów. Po negocjacji zasobów, w zależności od konfiguracji klastra, menadżer aplikacji może uruchomić zadanie w obecnym węźle lub przekazać poszczególne części zadania do innych węzłów roboczych. Na rysunku 3.2. przedstawiona została sytuacja, w której menadżer aplikacji podzielił zadanie na dwie części i przesłał po jednym z nich do oddzielnych węzłów roboczych.



Rys. 3.2. Działanie programu MapReduce na platformie Hadoop w modelu rozproszonym.

Po otrzymaniu żądania wykonania fragmentu zadania, menadżer węzła uruchamia proces wirtualnej maszyny Java oraz uruchamia na nim proces YARN, który odpowiedzialny będzie za wykonanie danego zadania, może to być zadanie mapujące, redukujące, łączące etc. Proces YARN rozpoczyna zadanie i czeka aż zostanie ono wykonane. Jak tylko zadanie zostanie wykonane, informacja o jego wykonaniu przekazywana jest do menadżera aplikacji, który monitoruje status wszystkich fragmentów zadania. Po wykonaniu wszystkich zadań, menadżer aplikacji kończy działanie programu oraz wysyła informację do menadżera zasobów.

W pracy została użyta maszyna wirtualna z platformą Hadoop udostępniana przez firmę Cloudera. Klaster na maszynie wirtualnej domyślnie skonfigurowany jest w modelu pseudo rozproszonym. Działanie zadania MapReduce w modelu pseudo rozproszonym zostało zaprezentowane na rysunku 3.3. Model ten imituje działanie klastra, tzn. różne procesy *daemony* Hadoopa wykonywane są na różnych instancjach maszyny wirtualnej Java. W tym modelu znajduje się tylko jeden menadżer węzła, który odpowiedzialny jest za zainicjowanie zarówno menadżera aplikacji jak i uruchomienie procesów YARN.



Rys. 3.3. Działanie aplikacji MapRedue w modelu pseudo-rozproszonym.

### 3.3. Implementacja programów na platformie Hadoop

Pierwszym krokiem do stworzenia programu MapReduce działającego na platformie Hadoop jest określenie kodu klasy mapującej, która musi rozszerzać wbudowaną klasę generyczną o następującej sygnaturze:

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
```

gdzie:

- *KEYIN* – typ klucza  $k_1$ ,
- *VALUEIN* – typ wartości  $v_1$  dla danego klucza  $k_1$ ,
- *KEYOUT* – typ wyjściowy klucza  $k_2$ ,
- *VALUEOUT* – typ wartości wyjściowej  $v_2$  dla danego klucza  $k_2$ .

Należy także przesłonić metodę *map*, której definicja została przedstawiona na listingu 3.1. Zdefiniowanie tej metody w programie użytkownika przesłoni jej domyślną implementację.

```
protected void map(KEYIN key, VALUEIN value, Context context) throws  
    IOException, InterruptedException {  
    context.write((KEYOUT) key, (VALUEOUT) value);  
}
```

Listing 3.1. Domyślna implementacja metody mapującej.

W przypadku kiedy użytkownik nie zdecyduje się na przesłonięcie tej metody, wartości zostaną przekazane do dalszej części programu (zapisane do kontekstu) w niezmienionym formacie tzn. dane postaci klucz-wartość będą jednakowe na wejściu jak i na wyjściu kroku mapującego. Dodatkowym parametrem jest obiekt klasy *Context*, który jest kontekstem całego programu. Do kontekstu należy zapisać wartości wyjściowe  $k_2$  oraz  $v_2$ , które będą przekazane do dalszej części procesu. Dodatkowo, użytkownik ma możliwość implementacji metod *setup* oraz *cleanup*, które wywoływane są dokładnie raz przed i po wykonaniu całego kroku mapującego. Implementacja tych metod jest użyteczna do ustawienia konfiguracji procesu mapującego lub usunięcia plików po jego zakończeniu. Sygnatury metod *setup* oraz *cleanup* są następujące:

```
protected void setup(Context context) throws IOException,  
    InterruptedException
```

```
protected void cleanup(Context context) throws IOException,  
    InterruptedException
```

Użytkownicy mają również możliwość definiować przebieg całego kroku mapującego poprzez przesłonięcie metody *run* przedstawionej na listingu 3.2. Metoda ta jest odpowiedzialna za wykonanie logiki mapującej dla każdej pary klucz-wartość przekazanej z kontekstu do programu [8].

```
public void run(Context context) throws IOException, InterruptedException
{
    setup(context);
    try { while (context.nextKeyValue()) {
        map(context.getCurrentKey(), context.getCurrentValue(),
            context);
    }
    } finally { cleanup(context); }
}
```

Listing 3.2. Domyślna implementacja logiki uruchomienia kroku mapującego.

Po definicji klasy mapującej można przejść do implementacji klasy redukującej. Klasa redukująca powinna dziedziczyć po klasie generycznej *Reducer*:

```
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT>
```

gdzie:

- *KEYIN* - typ klucza  $k_2$ ,
- *VALUEIN* - typ wartości  $v_2$  ze zbioru  $list(v_2)$ ,
- *KEYOUT* - typ wyjściowy klucza  $k_3$ ,
- *VALUEOUT* - typ wyjściowy wartości  $v_3$ .

Należy również dokonać implementacji metody *reduce*, która przedstawiona została na listingu 3.3. Tak jak w przypadku klasy mapującej, metoda ta posiada domyślną implementację. Dodatkowo, klucz  $k_3$  oraz wartość klucza  $v_3$  należy zapisać do kontekstu aby przekazać go do dalszej części programu.

```
protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context
    context) throws IOException, InterruptedException {
    for(VALUEIN value: values) {
        context.write((KEYOUT) key, (VALUEOUT) value);
    }
}
```

Listing 3.3. Domyślna implementacja metody redukującej.

Podobnie jak w przypadku klasy mapującej, użytkownik ma możliwość przesłonięcia metod *setup* oraz *cleanup*:

```
protected void setup(Context context
                    ) throws IOException, InterruptedException {
}

protected void cleanup(Context context
                       ) throws IOException, InterruptedException {
}
```

Możliwe jest również przesłonięcie metody odpowiedzialnej za wykonanie logiki redukującej. Implementacja domyślna tej metody została przedstawiona na listingu 3.4.

```
public void run(Context context) throws IOException, InterruptedException
{
    setup(context);
    try { while (context.nextKey()) {
        reduce(context.getCurrentKey(), context.getValues(), context);
    }
    } finally { cleanup(context); }
}
```

Listing 3.4. Domyślna implementacja logiki uruchomienia kroku redukującego.

Zdefiniowanie klasy mapującej jest konieczne do poprawnego działania programu MapReduce. Jeśli użytkownik nie dokona implementacji własnej logiki dla klasy mapującej, użyta będzie implementacja domyślna. Zdefiniowanie klasy redukującej nie jest konieczne do poprawnego wykonania programu. Przykładem jest zapis do bazy HBase, gdzie w przypadku braku implementacji klasy *Reducer*, program przekaże wartości wyjściowe z kroku mapującego na wyjście programu – zapisze do bazy. Dodatkowo, dodanie informacji o klasach odpowiedzialnych za łączenie oraz partycjonowanie jest opcjonalne. W niektórych przypadkach, różne klasy działające na wyższym poziomie abstrakcji, np. klasy użytkowe służące do zapisu danych do bazy HBase, oferują implementację partycjonowania lub łączenia na podstawie stanu innych elementów ekosystemu Hadoop. W takim przypadku, użytkownik nie musi sam pisać implementacji klasy partycjonującej, wystarczy, że odpowiednio wykorzysta dostępne klasy oraz metody biblioteki HBase.

Klasa odpowiedzialna za partycjonowanie musi dziedziczyć po klasie *Partitioner* oraz przesyłać metodę *getPartition*:

```
public abstract class Partitioner<KEY,VALUE>{

    public abstract int getPartition(KEY key, VALUE value, int
    numReduceTasks)
}
```

gdzie:

- *KEY* - typ klucza,
- *VALUE* - typ wartości dla danego klucza,
- *numReduceTasks* - liczba zadań redukujących przypisanych dla całego procesu.

Metoda ta zwraca zmienną *numReduceTasks*, która informuje o tym, w której partycji powinna znaleźć się dana para klucz-wartość. W przypadku operacji łączenia, dana klasa musi dziedziczyć po klasie *Reducer*, tak samo jak w przypadku klasy redukującej. Zdefiniowana jest inaczej w klasie głównej programu. Klasy *Mapper* oraz *Reducer* posiadają także abstrakcyjne klasy wewnętrzne o sygnaturach odpowiednio:

```
public abstract class Context
implements MapContext<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {
}

public abstract class Context
implements ReduceContext<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {
}
```

Program MapReduce odpowiedzialny jest za stworzenie oraz przekazanie kontekstu programu do aplikacji. Kontekst aplikacji służy głównie do zapisywania oraz przekazywania rezultatów z poszczególnych kroków programu MapReduce.

### **3.4. Instalacja oraz weryfikacja środowiska uruchomieniowego**

Do wykonania wszystkich programów w dalszej części pracy została wykorzystana platforma Hadoop dystrybucji firmy Cloudera. Platforma jest dostarczana w formie obrazu systemu operacyjnego Linux wraz z skonfigurowanym ekosystemem Hadoop. System ten może zostać zainstalowany na dowolnej maszynie wirtualnej wspieranej w danej chwili przez firmę Cloudera. W pracy została wykorzystana maszyna wirtualna firmy VMWare. Obraz

systemu *Cloudera QuickStart for CDH* w najnowszej wersji można pobrać z oficjalnej strony firmy: [https://www.cloudera.com/downloads/quickstart\\_vms/5-13.html](https://www.cloudera.com/downloads/quickstart_vms/5-13.html).

Aby poprawnie korzystać ze środowiska Hadoop należy dokonać weryfikacji i konfiguracji następujących plików XML: `core-site.xml`, `hdfs-site.xml`, `yarn-site.xml` oraz `mapred-site.xml`. Pliki konfiguracyjne znajdują się w katalogu `/etc/hadoop/conf` systemu operacyjnego Linux. Minimum konfiguracyjne dla pliku `core-site.xml` znajduje się na listingu 3.5. Wpis ten ustawia domyślny system plików w klastrze. Na listingu 3.6. znajduje się fragment pliku `hdfs-site.xml`, który określa ilość replik jakie zostaną stworzone dla każdego bloku danych w systemie. Jeśli jedna z replik jest niedostępna wtedy Hadoop będzie mógł pobrać dane z innego węzła danych. Dodatkowo, plik ten zawiera informacje o węzłach przestrzeni nazw lub adresach serwisów webowych.

```
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://quickstart.cloudera:8020</value>
</property>
```

Listing 3.5. Definicja domyślnego systemu plików używanego w klastrze.

```
<property>
  <name>dfs.replication</name>
  <value>3</value>
</property>

<property>
  <name>dfs.datanode.address</name>
  <value>0.0.0.0:50010</value>
</property>
<property>
  <name>dfs.namenode.http-address</name>
  <value>0.0.0.0:50070</value>
</property>
```

Listing 3.6. Fragment pliku `hdfs-site.xml`.

Na listingu 3.7. przedstawiony jest fragment pliku `yarn-site.xml`, który pozwala zdefiniować rodzaj planisty zasobów oraz podstawowe dane konfiguracyjne dla programów wykonywanych w klastrze, np. biblioteki dla programów pracujących w systemie YARN. Kluczowym plikiem konfiguracyjnym dla programów MapReduce jest przedstawiony na listingu 3.8. plik `mapred-site.xml`. Informuje on klaster m.in. o tym, że wszystkie programy MapReduce muszą być wykonywane przez planistę zasobów i przechodzić przez system YARN. Dzięki temu programy MapReduce mają gwarancję optymalnego wykorzystania zasobów dostępnych w klastrze.

```

<property>
  <name>yarn.resourcemanager.scheduler.class</name>
  <value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.
    capacity.CapacityScheduler</value>
</property>
<property>
  <description>Classpath for typical applications.</description>
  <name>yarn.application.classpath</name>
  <value>
    $HADOOP_CONF_DIR,
    $HADOOP_COMMON_HOME/*,$HADOOP_COMMON_HOME/lib/*,
    $HADOOP_HDFS_HOME/*,$HADOOP_HDFS_HOME/lib/*,
    $HADOOP_MAPRED_HOME/*,$HADOOP_MAPRED_HOME/lib/*,
    $HADOOP_YARN_HOME/*,$HADOOP_YARN_HOME/lib/*
  </value>
</property>

```

Listing 3.7. Fragment pliku yarn-site.xml odpowiedzialnego za działanie systemu YARN oraz definicję planisty zasobów.

```

<property>
  <name>mapreduce.framework.name</name>
  <value>yarn</value>
</property>

```

Listing 3.8. Minimum konfiguracyjne dla pliku mapred-site.xml.

Konfiguracja wszystkich powyższych plików wraz z wyróżnionymi elementami pozwala na optymalne uruchomienie programów MapReduce na platformie Hadoop.

## 4. Przetwarzanie i analiza danych przy użyciu MapReduce

### 4.1. Analiza plików tekstowych

#### Definicja danych i kodu programu

W celu pokazaniu działania oraz analizy procesu MapReduce na platformie Hadoop został napisany przykładowy program analizujący 4 pliki tekstowe pobrane z systemu HDFS o podobnej strukturze oraz zapisujący rezultaty do nowego pliku, utworzonego w trakcie procesu. Każdy z plików tekstowych zawiera identyfikator z przedziału <1,6>, myślnik oraz 8 cyfr z zakresu <0,9>. Kilka pierwszych rekordów z pliku tekstowego pokazano poniżej.

2-22586534
3-58614618
4-04430583
5-85054397
6-78171930
1-27687382
2-79253991
3-81106596
...

W tym przykładzie rezultatem funkcji mapującej ma być zestaw danych  $(k_2, v_2)$ , w którym  $k_2$  oznacza numer identyfikacyjny zaś  $v_2$  oznacza wartość sześciu ostatnich cyfr danego wiersza. Funkcja redukująca będzie polegała na znalezieniu maksymalnej wartości  $v_2$  dla każdego  $k_2$ . Działanie programu można przedstawić przy pomocy poniższej formuły:

$$\begin{aligned} \text{Map}(k_1, v_1) &\rightarrow \text{list}(k_2, v_2) \\ \text{Reduce}(k_2, \text{list}(v_2)) &\rightarrow \text{list}(k_2, \max(\text{list}(v_2))) \end{aligned}$$

W wyniku działania programu spodziewać się należy sześciu wyników z maksymalną wartością dla każdego numeru identyfikacyjnego. Kod klasy mapującej przedstawiono na listingu 4.1 zaś kod klasy redukującej na listingu 4.2.

```
public class MaxValueMapper extends Mapper<LongWritable, Text, Text,
IntWritable> {

    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException
    {
```

```

        String line = value.toString();
        String id;
        int lineValue;
        id = line.substring(0, 1);
        lineValue = Integer.parseInt(line.substring(4,10));
        context.write(new Text(id), new IntWritable(lineValue));
    }
}

```

Listing 4.1. Kod klasy mapującej.

```

public class MaxValueReducer
extends Reducer<Text, IntWritable, Text, IntWritable>
{
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context
context)
    throws IOException, InterruptedException
    {
        int maxValue = Integer.MIN_VALUE;
        for(IntWritable value : values){
            maxValue = Math.max(maxValue, value.get());
        }
        context.write(key, new IntWritable(maxValue));
    }
}

```

Listing 4.2. Kod klasy redukującej.

Ostatnim etapem jest utworzenie klasy głównej programu, w której nastąpi uruchomienie klasy mapującej i redukującej a także zainicjowanie danych wejściowych. Do analizy wykorzystanych zostały cztery pliki tekstowe o łącznej wielkości około 420 MB danych. Kod główny programu znajduje się na listingu 4.3.

```

public class MaxValue {
    public static void main(String... args) throws Exception
    {

        Job job = new Job();
        job.setJarByClass(MaxValue.class);
        job.setJobName("Maksymalna wartosc dla plikow z
logami.");

        FileInputFormat.addInputPath(job, new
Path("/user/cloudera/logfile1.txt"));
        FileInputFormat.addInputPath(job, new
Path("/user/cloudera/logfile2.txt"));
        FileInputFormat.addInputPath(job, new
Path("/user/cloudera/logfile3.txt"));
        FileInputFormat.addInputPath(job, new
Path("/user/cloudera/logfile4.txt"));
        FileOutputFormat.setOutputPath(job, new
Path("/user/cloudera/output"));
    }
}

```

```

        job.setMapperClass(MaxValueMapper.class);
        job.setReducerClass(MaxValueReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}

```

Listing 4.3. Kod funkcji głównej programu MapReduce.

W funkcji *main* przekazane zostały do programu informacje o danych znajdujących się w systemie HDFS a także katalog, do którego zapisane zostaną rezultaty obliczeń. Aby program MapReduce zadziałał poprawnie należy przekazać informacje o typach danych wejściowych oraz wyjściowych z kroków *map* oraz *reduce*.

### Uruchomienie programu oraz analiza rezultatów

Przed uruchomieniem programu należy upewnić się, że wszystkie wskazane pliki znajdują się w systemie HDFS. Wykonując komendy z listingu 4.4. z katalogu, w którym znajdują się wszystkie pliki potrzebne do analizy, utworzone zostaną odpowiednie pliki w systemie HDFS. Listing 4.5. zawiera komendy, które służą do uruchomienia programu w klastrze.

```

hadoop fs -copyFromLocal logfile1.txt /user/cloudera/logfile1.txt
hadoop fs -copyFromLocal logfile2.txt /user/cloudera/logfile2.txt
hadoop fs -copyFromLocal logfile3.txt /user/cloudera/logfile3.txt
hadoop fs -copyFromLocal logfile4.txt /user/cloudera/logfile4.txt

```

Listing 4.4. Komendy służące do przekazania danych do system HDFS.

```

export HADOOP_CLASSPATH=$(hadoop classpath)
javac -classpath ${HADOOP_CLASSPATH} *.java
jar -cvf MaxValue.jar *.class
hadoop jar MaxValue.jar MaxValue

```

Listing 4.5. Tworzenie oraz uruchomienie pliku jar zawierającego zadanie MapReduce.

Po wykonaniu komend z listingów 4.4. oraz 4.5. zostanie uruchomiony program MapReduce. Na rysunku 4.1. widać start programu, w pierwszej kolejności program łączy się z zarządcą zasobów, aby przekazać informacje o zadaniu. Liczba ścieżek, z których należy pobrać dane *total input paths to process* jest równa 4, wynika to z fakty posiadania czterech plików wejściowych. Kolejnym istotnym elementem jest liczba podziałów *number of splits*, która również wynosi 4. Jest to liczba oznaczająca, jak wiele partycji należy odczytać, aby

pobrać dane. Jako, że pliki użyte w zadaniu są mniejsze od 128MB (domyślna wielkość bloku w systemie HDFS) to każdy plik mieści się na jednej partycji, wobec tego liczba podziałów jest równa 4. Pozostałe wpisy oznaczają poprawność zainicjowania programu, przydział identyfikatora zadania oraz adres URL, pod którym można śledzić postęp zadania.

```
[cloudera@quickstart sourcecode2]$ hadoop jar MaxValue.jar MaxValue
18/03/22 14:19:23 INFO client.RMPProxy: Connecting to ResourceManager at /0.0.0.0:8032
18/03/22 14:19:24 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement the Tool interface and execute your application with ToolRunner to remedy this.
18/03/22 14:19:24 INFO input.FileInputFormat: Total input paths to process : 4
18/03/22 14:19:24 INFO mapreduce.JobSubmitter: number of splits:4
18/03/22 14:19:25 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1521749203366_0001
18/03/22 14:19:26 INFO impl.YarnClientImpl: Submitted application application_1521749203366_0001
18/03/22 14:19:26 INFO mapreduce.Job: The url to track the job: http://quickstart.cloudera:8088/proxy/application_1521749203366_0001/
18/03/22 14:19:26 INFO mapreduce.Job: Running job: job_1521749203366_0001
18/03/22 14:19:38 INFO mapreduce.Job: Job job_1521749203366_0001 running in uber mode : false
18/03/22 14:19:39 INFO mapreduce.Job: map 0% reduce 0%
18/03/22 14:21:33 INFO mapreduce.Job: map 1% reduce 0%
18/03/22 14:21:36 INFO mapreduce.Job: map 2% reduce 0%
18/03/22 14:21:37 INFO mapreduce.Job: map 3% reduce 0%
```

Rys. 4.1. Start programu MapReduce.

Powyższy program został uruchomiony w modelu pseudo rozproszonym czyli węzeł klienta, nadzorcy zasobów a także węzły robocze znajdują się na jednej maszynie. Do realizacji tego zadania zostały przydzielone 2 wirtualne procesory oraz 4GB RAM. Po skończonym przetwarzaniu Hadoop wyświetla statystyki odnośnie programu. Kluczowe informacje jakie można w ten sposób otrzymać to np. wielkość pobranych oraz zapisanych danych. Statystyki wykonania tego programu przedstawione zostały na rysunku 4.2.

```
HDFS: Number of bytes read=440000492
HDFS: Number of bytes written=54
HDFS: Number of read operations=15
HDFS: Number of large read operations=0
HDFS: Number of write operations=2
```

Rys. 4.2. Statystyki po skończeniu programu MapReduce.

Program MapReduce wczytał 440000492 bajtów danych (około 420MB) oraz zapisał 54 bajty danych do pliku wyjściowego zawierającego wyniki przetwarzania. Wyniki znajdują się w jednym z plików w systemie HDFS, rezultaty zostały przedstawione na rysunku 4.3.

```
[cloudera@quickstart sourcecode2]$ hadoop fs -cat /user/cloudera/output/part-r-00000
1      999999
2      999999
3      999999
4      999999
5      999999
6      999999
```

Rys. 4.3. Rezultat wykonania obliczeń.

Dla każdego identyfikatora znaleziono wartość maksymalną równą 999999. Dodatkowo, wchodząc pod wskazany na początku uruchomienia programu adres URL otrzymamy podsumowanie statystyk dotyczących danego zadania. Rysunek 4.4. przedstawia zrzut ekranu logów udostępnianych przez graficzny interfejs stworzony dla logów systemu YARN. Jak widać na rysunku, program został wykonany w 8 minut i 21 sekundy, zostały przydzielone 4 zadania mapujące (po jednym do każdego z plików) a także jedno zadanie redukcyjne.

<b>Job Name:</b>	Maksymalna wartosc dla plikow z logami.
<b>User Name:</b>	cloudera
<b>Queue:</b>	root.cloudera
<b>State:</b>	SUCCEEDED
<b>Uberized:</b>	false
<b>Submitted:</b>	Thu Mar 22 14:19:25 PDT 2018
<b>Started:</b>	Thu Mar 22 14:19:37 PDT 2018
<b>Finished:</b>	Thu Mar 22 14:27:58 PDT 2018
<b>Elapsed:</b>	8mins, 21sec
<b>Diagnostics:</b>	
<b>Average Map Time</b>	6mins, 51sec
<b>Average Shuffle Time</b>	21sec
<b>Average Merge Time</b>	24sec
<b>Average Reduce Time</b>	30sec

Rys. 4.4. Podsumowanie zadania w interfejsie graficznym.

### Porównanie rezultatów z programem analizującym dane w sposób sekwencyjny

Dla sprawdzenia poprawności przeprowadzonej analizy wykonano również skrypt powłoki linuxowej w języku Bash, który realizuje dokładnie to samo zadanie w sposób sekwencyjny. Kod skryptu został przedstawiony na listingu 4.6.

```
#!/bin/bash
now="$(date) "
echo "$now"
filename="$1"
max_val=0
max_num=0
tablica=(0 0 0 0 0 0)

while read -r line
do
```

```

num="${line:0:1}"
value="${line:3:6}"
if [ "$value" -gt "${tablica[num-1]}" ]
then
    tablica[num-1]="$value"
    max_num="$num"
fi

done < "$1"
now="$(date)"
echo "$now"
echo "1 = ${tablica[0]}"
2 = ${tablica[1]}
3 = ${tablica[2]}
4 = ${tablica[3]}
5 = ${tablica[4]}
6 = ${tablica[5]}"

```

Listing 4.6. Skrypt powłoki realizujący zadanie wykonane poprzednio przez program MapReduce.

Wszystkie pliki użyte w programie MapReduce zostały połączone w jeden plik *resfile.txt*. Otrzymano identyczne rezultaty jak w przypadku programu MapReduce. Przy uruchomieniu programu na tej samej maszynie czas analizy pliku wynosił około 22 minuty – 3 razy dłużej niż w przypadku MapReduce.

## 4.2. Łączenie danych pochodzących z plików tekstowych oraz bazy NoSQL

### Przygotowanie danych wejściowych oraz kodu źródłowego

Algorytm MapReduce nie narzuca rodzaju lub formatu danych wejściowych. Wszystkie rodzaje: pliki, rekordy z bazy SQL, NoSQL, Webservice są dozwolone na wejściu konkretnego zadania MapReduce, o tyle, o ile dane wyjściowe z kroku *Map* są jednakowe i możliwe do dalszej analizy. W tym rozdziale został zaprezentowany przykład procesu MapReduce, który pobiera dane jednocześnie z bazy HBase oraz dwóch plików tekstowych. Dodatkowo, zaimplementowana została klasa dziedzicząca po klasie *Partitioner*, która umożliwia rozbięcie danych wyjściowych z kroku *Map* na partycje i zaaplikowanie kroku *Reduce* dla poszczególnych partycji. Wynikiem tej operacji są 2 pliki tekstowe reprezentujące rezultaty obliczeń dla konkretnych partycji. Zaprezentowany został również przykład użycia kroku łączącego *Combiner*. Stosuje się ją po to, aby logikę redukującą zastosować na maszynach wykonujących zadania mapujące w celu zmniejszenia ilości

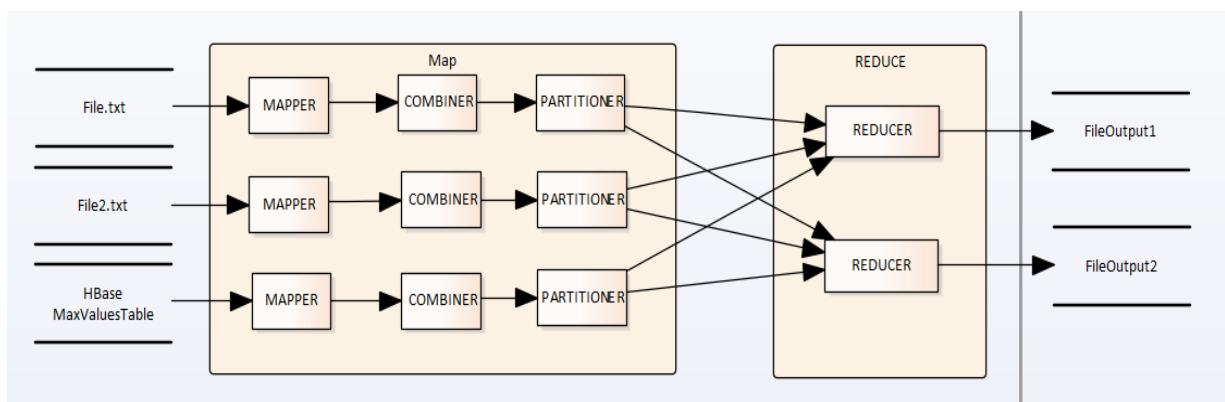
danych przesyłanych pomiędzy węzłami. Do tego przykładu użyto plików tekstowych o strukturze podobnej do plików z podrozdziału 4.1.

Ponownie, dla każdego identyfikatora pobieramy 6 ostatnich cyfr oraz obliczamy maksymalną wartość. Tym razem pliki wejściowe posiadają 10GB danych, zwiększone zostały również zasoby obliczeniowe maszyny do odpowiednio 13,4GB pamięci RAM oraz 4 wirtualnych procesorów. Dodatkowo, do obliczeń została dodana tabela o strukturze pokazanej na rysunku 4.5.

```
hbase(main):007:0> scan 'MaxValuesTable'
ROW          COLUMN+CELL
1            column=values:first_value, timestamp=1527375811116, value=3463
2            column=values:first_value, timestamp=1527375820595, value=869533
3            column=values:first_value, timestamp=1529270074661, value=10999123
4            column=values:first_value, timestamp=1527375830109, value=222333
5            column=values:first_value, timestamp=1527375790787, value=505439
6            column=values:first_value, timestamp=1527375790787, value=817193
6 row(s) in 0.0250 seconds
```

Rysunek 4.5. Struktura tabeli MaxValuesTable w bazie HBase.

Dla każdego rekordu mamy klucz *ROW* odpowiadający identyfikatorowi z pliku tekstowego, jedną rodzinę kolumn *values*, kolumnę *first\_value*, oraz wartość, która zostanie pobrana od obliczeń. Celem zadania jest połączenie danych z pliku tekstowego i bazy oraz wywnioskowanie maksymalnej wartości dla danego identyfikatora. W bazie jeden z rekordów celowo posiada wartość większą niż 999999, aby pokazać, że dane w bazie rzeczywiście brane są pod uwagę podczas analizy. Przebieg takiego procesu można scharakteryzować używając diagramu przedstawionego na rysunku 4.6.



Rysunek 4.6. Schemat przebiegu zadania MapReduce dla różnych danych wejściowych tworzonego.

Na listingach 4.7 oraz 4.8. pokazano odpowiednio klasy mapujące dla plików tekstowych oraz tabeli z bazy HBase.

```
public class MaxValueMultipleInputFileMapper extends
    Mapper<LongWritable, Text, Text, IntWritable> {

    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String line = value.toString();
        String id;
        int lineValue;
        id = line.substring(0, 1);
        lineValue = Integer.parseInt(line.substring(4, 10));
        context.write(new Text(id), new IntWritable(lineValue));
    }
}
```

Listing 4.7. Klasa mapująca dla plików tekstowych.

```
public class MaxValueMultipleInputHBaseMapper extends
    TableMapper<Text, IntWritable> {

    @Override
    public void map(ImmutableBytesWritable key, Result values,
        Context context)
        throws IOException, InterruptedException {
        String value = new
            String(values.getValue(Bytes.toBytes("values"),
                Bytes.toBytes("first_value")));
        context.write(new Text(key.get()),
            new IntWritable(Integer.valueOf(value)));
    }
}
```

Listing 4.8. Klasa mapująca dla tabeli HBase.

Platforma Hadoop udostępnia odpowiednie API do tworzenia połączeń z bazą HBase. W tym przypadku klasa *MaxValueMultipleInputHBaseMapper* dziedziczy po klasie *TableMapper* z pakietu *org.apache.hadoop.hbase* o następującej sygnaturze:

```
public abstract class TableMapper<KEYOUT, VALUEOUT> extends
    Mapper<ImmutableBytesWritable, Result, KEYOUT, VALUEOUT>
```

Jak widać na podstawie powyższej definicji, klasa *TableMapper* dziedziczy po klasie *Mapper*, która została użyta w pierwszym przykładzie, dlatego możliwe jest użycie jej w programie jako klasy mapującej. Na listingu 4.9. znajduje się kod klasy odpowiedzialnej za partycjonowanie wyników z kroku mapującego.

```

public class MaxValueMultipleInputPartitioner extends
    Partitioner<Text,IntWritable>{
    @Override
    public int getPartition(Text key, IntWritable value, int
        numReduceTasks) {
        if(Integer.valueOf(key.toString())>=4){
            return 0;
        }
        else {
            return 1;
        }
    }
}

```

Listing 4.9. Klasa odpowiedzialna za partycjonowanie rezultatów przed przesłaniem do redukcji.

Biorąc pod uwagę, że wartości identyfikatora są z przedziału <1,6> użyte zostały 2 partycje, gdzie wartości <1,3> trafiają do partycji pierwszej natomiast wartości <4,6> trafiają do partycji drugiej. Na listingu 4.10. znajduje się kod klasy redukującej.

```

public class MaxValueMultipleInputReducer extends
    Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context
        context)
        throws IOException, InterruptedException {
        int maxValue = Integer.MIN_VALUE;
        for (IntWritable value : values) {
            maxValue = Math.max(maxValue, value.get());
        }
        context.write(key, new IntWritable(maxValue));
    }
}

```

Listing 4.10. Klasa odpowiedzialna za partycjonowanie rezultatów przed przesłaniem do redukcji.

Ta sama klasa redukująca jest aplikowana dla wszystkich partycji. Listing 4.11. zawiera kod głównej klasy programu.

```

public class MaxValueMultipleInput {

    public static void main(String[] args) throws IOException,
        ClassNotFoundException, InterruptedException {

        Scan scan = new Scan();
        scan.setCaching(500);
        scan.setCacheBlocks(false);

        Job job = new Job();
        job.setJarByClass(MaxValueMultipleInput.class);
        job.setJobName("Maksymalna wartosc dla plikow z logami.");

        TableMapReduceUtil.initTableMapperJob("MaxValuesTable", scan,

```

```

        MaxValueMultipleInputHBaseMapper.class, Text.class,
        IntWritable.class, job);

        MultipleInputs.addInputPath(job, new Path("file.txt"),
        TextInputFormat.class, MaxValueMultipleInputFileMapper.class);

        MultipleInputs.addInputPath(job, new Path("file2.txt"),
        TextInputFormat.class, MaxValueMultipleInputFileMapper.class);

        MultipleInputs.addInputPath(job, new Path("not needed"),
        TableInputFormat.class,
        MaxValueMultipleInputHBaseMapper.class);

        FileOutputFormat.setOutputPath(job, new Path("output"));

        job.setNumReduceTasks(2);

        job.setPartitionerClass(MaxValueMultipleInputPartitioner.class)
        ;
        job.setCombinerClass(MaxValueMultipleInputReducer.class);
        job.setReducerClass(MaxValueMultipleInputReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}

```

Listing 4.11. Klasa główna programu.

Utworzona została instancja klasy “Scan”, która odpowiada za konfigurację sposobu pobierania danych z bazy, w tym przypadku głównie za buforowanie danych. Przekazywana jest jako parametr w późniejszym etapie programu. Do kontekstu programu została dodana informacja o tabeli wejściowej z bazy HBase poprzez wywołanie metody: *TableMapReduceUtil.initTableMapperJob*. Kluczowym elementem programu było wywołanie metody *MultipleInputs.addInputPath*, która odpowiada za dodanie informacji o wszystkich danych wejściowych do programu MapReduce. Jej sygnatura jest następująca:

```

public static void addInputPath(JobConf conf, Path path,
                               Class<? extends InputFormat> inputFormatClass,
                               Class<? extends Mapper> mapperClass)

```

Gdzie:

1. conf - konfiguracja programu
2. path - ścieżka, która zostaje dodana do pliku, należy tutaj zauważyć, że w przypadku dodania pliku tekstowego parametr ten jest konieczny, ale w przypadku dodania do programu informacji o tabelach z bazy HBase ten parametr nie ma znaczenia i jest pomijany

3. InputFormatClass - typ formatu dla danych wejściowych
4. mapperClass - klasa mapująca dla danych wejściowych, jedna klasa może być użyta w przypadku danych wejściowej o podobnej strukturze.

Za właściwe dodanie klasy łączącej i klasy odpowiadającej za partycjonowanie rezultatów odpowiada następujący fragment kodu:

```
job.setPartitionerClass(MaxValueMultipleInputPartitioner.class);  
job.setCombinerClass(MaxValueMultipleInputReducer.class);
```

### Uruchomienie programu oraz analiza rezultatów

Aby uruchomić program poprawnie należy upewnić się, że oba pliki /user/cloudera/input\_data/File.txt oraz /user/cloudera/input\_data/File2.txt znajdują się w systemie HDFS. Następnie należy wygenerować plik .jar zawierający kod źródłowy program. Możliwe jest także, że wykonanie programu w klastrze będzie wymagało dodanie dodatkowych bibliotek bazy HBase do zmiennej HADOOP\_CLASSPATH. Dodanie odpowiednich bibliotek do zmiennej środowiskowej znajduje się na listingu 4.12.

```
export HADOOP_CLASSPATH=$(hadoop classpath)  
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hbase/lib/*  
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hbase/*
```

Listing 4.12. Załączanie bibliotek HBase do zmiennej HADOOP\_CLASSPATH.

Dzienniki z wykonania programu zostały przedstawione na listingu 4.13. Całkowity czas wykonania programu to 13 minut oraz 28 sekund. Przeciętny czas każdego zadania mapującego to 21 sekund natomiast wykonanych zostało aż 83 zadania mapujące pomimo tego, że program potrzebował przeprowadzić obliczenia na jedynie 2 plikach. Wynika to z faktu, że liczba danych w plikach tekstowych wynosiła około 10GB, dodatkowo program potrzebował przeczytać dane z bazy NoSQL, natomiast blok w systemie HDFS jest domyślnie równy 128MB. Każdy blok danych otrzymuje jedno zadanie mapujące, dlatego liczba zadań mapujących jest tak duża. Liczba zadań redukujących wyniosła 2 – zgodnie z parametrem ustawionym w programie (1 reduktor dla partycji).

```

18/05/26 13:35:03 INFO zookeeper.ZooKeeper: Initiating client connection, connectString=localhost:2181 sessionTimeout=90000
watcher=hconnection-0x3872bb090x0, quorum=localhost:2181, baseZNode=/hbase
18/05/26 13:35:03 INFO zookeeper.ClientCnxn: Opening socket connection to server quickstart.cloudera/127.0.0.1:2181. Will not attempt
to authenticate using SASL (unknown error)
18/05/26 13:35:03 INFO zookeeper.ClientCnxn: Socket connection established, initiating session, client: /127.0.0.1:32998, server:
quickstart.cloudera/127.0.0.1:2181
18/05/26 13:35:04 INFO zookeeper.ClientCnxn: Session establishment complete on server quickstart.cloudera/127.0.0.1:2181, sessionId =
0x1640dacedce0020, negotiated timeout = 40000
18/05/26 13:35:04 INFO util.RegionSizeCalculator: Calculating region sizes for table "MaxValuesTable" [...]
[...]
18/05/26 13:35:05 INFO input.FileInputFormat: Total input paths to process : 2
[...]
18/05/26 13:35:06 INFO mapreduce.JobSubmitter: number of splits:83
[...]
18/05/26 13:35:16 INFO mapreduce.Job: map 0% reduce 0%
[...]
18/05/26 13:48:30 INFO mapreduce.Job: map 100% reduce 100%

```

Listing 4.13. Dzienniki zadania MapReduce.

Dodatkowo, w dziennikach pojawiła się informacja odnośnie komunikacji z bazą HBase. Widoczne są szczegóły połączenia, informacje o uwierzytelnianiu oraz tworzeniu sesji użytkownika. Podobnie jak w przypadku plików tekstowych MapReduce sprawdza tabelę źródłową w celu oszacowania jej wielkości oraz przydzielenia odpowiedniej liczby zadań mapujących. Wynikiem powyższego programu są dwa pliki tekstowe (po jednym pliku dla każdej partycji): part-r-0000 oraz part-r-0001, które znajdują się w systemie HDFS. Rezultaty obliczeń zostały przedstawione na listingu 4.14. Widać na nim, że jeden z rekordów ma wartość większą niż 999999, jest to wartość pochodząca z bazy HBase.

```

[cloudera@quickstart 2nd analysis]$ hadoop fs -cat
/user/cloudera/output/part-r-00000
4      999999
5      999999
6      999999
[cloudera@quickstart 2nd analysis]$ hadoop fs -cat
/user/cloudera/output/part-r-00001
1      999999
2      999999
3      10999123

```

Listing 4.14. Rezultaty obliczeń drugiej analizy z wykorzystaniem MapReduce.

Na podstawie działania powyższego programu można zauważyć kilka charakterystycznych elementów programów działających w modelu MapReduce. W porównaniu do programu z podrozdziału 4.1. czas wykonania był 1.5 razy dłuższy ale wielkość danych była ponad 20-krotnie większa. Nie zostały dokonane żadne zmiany

konfiguracyjne, ale zwiększone zostały zasoby maszyny wirtualnej. Wynika z tego, że algorytmy systemu YARN, z których korzystają programy MapReduce, dostosowują się automatycznie do ilości zasobów klastra. Dodatkowo, Program MapReduce doskonale radzi sobie z różnymi typami danych wejściowych, mogą to być pliki płaskie, rekordy z bazy SQL, NoSQL lub też dowolny format zdefiniowany przez użytkownika.

### 4.3. Zapis danych do bazy NoSQL

#### Konfiguracja bazy oraz przygotowanie kodu źródłowego

Kolejnym istotnym elementem modelu MapReduce jest fakt, że nie definiuje on formatu danych wyjściowych. Efektem wykonania programu MapReduce może być nie tylko plik płaski ale również stworzenie rekordów w bazie SQL lub NoSQL. W poniższym przykładzie do bazy danych HBase zostanie zapisanych 1GB danych pochodzących z plików tekstowych. Istotnym elementem w tym wypadku jest określenie odpowiednich regionów tabeli HBase aby zoptymalizować zapis danych. Listing 4.15. zawiera skrypt do utworzenia tabeli *MaxValuesRegion*, a także informacje o utworzonych regionach. Tabela zawiera trzy główne regiony, pierwszy gdzie klucz jest mniejszy (porównanie następuje przy użyciu porządku naturalnego) od 'H', drugi gdzie klucz jest równy lub większy od 'H' ale mniejszy od 'Q', oraz ostatni region, gdzie klucz jest większy lub równy 'Q'.

```
create 'MaxValuesRegion', 'values', {SPLITS => ['H','Q']}  
  
scan 'hbase:meta',{FILTER=>"PrefixFilter('MaxValuesRegion')"}
```

Listing 4.15. Tworzenie oraz prezentacja regionów w tabeli HBase.

W przypadku klastrów produkcyjnych różne region mogą znajdować się na różnych fizycznych maszynach. Do bazy w poniższym programie zostaną zapisane 2 pliki tekstowe o następującej strukturze:

```
XUGZMEOHXWFFY-13516357  
DCXXMMCFPVOY-18970434  
KBDYQXCVDTHB-84686245  
FSLVQQHNKYOX-89170200  
BPKZYUZVLEGN-99743499  
VRAFDNPKMLPU-20884250  
HURUCSYPVNNV-60590683  
KGVLEHWDJCOE-50586594
```

Ponownie, wartość przed „-” jest kluczem w modelu MapReduce, stanowić będzie również klucz rekordu w bazie HBase. Ostatnie sześć cyfr jest wartością, która zostanie

zapisana wraz z kluczem do bazy. Model MapReduce nie definiuje formatu danych wyjściowych, co więcej, do poprawnego zaimplementowania programu MapReduce na platformie Hadoop wcale nie potrzeba definiować klasy redukującej! W takim przypadku dane wychodzące z kroku Map są od razu przekazywane jako rezultat wykonania programu. Wynika to z faktu, że MapReduce zastosuje domyślną implementację funkcji redukującej. Na listingu 4.16. przedstawiony został kod klasy mapującej.

```
public class MaxValueMultipleInputFileMapper extends
    Mapper<LongWritable, Text, ImmutableBytesWritable, KeyValue> {

    @Override
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        String line = value.toString();
        String id = line.substring(0, 12);
        String number = line.substring(15, 21);

        ImmutableBytesWritable hkey = new ImmutableBytesWritable();

        hkey.set(id.getBytes());
        KeyValue kv = new KeyValue(hkey.get(), Bytes.toBytes("values"),

        Bytes.toBytes("first_value"),
        Bytes.toBytes(number));

        context.write(hkey, kv);
    }
}
```

Listing 4.16. Kod klasy mapującej przygotowujący rekordy do zapisu do bazy.

W klasie mapującej należy zwrócić uwagę, że oprócz parsowania linii z pliku wejściowego, metoda map tworzy także instancje obiektu `ImmutableBytesWritable`, która posłuży do zapisania klucza oraz do odpowiedniego czytania plików systemu HDFS zawierającego informacje o partycjach w tabeli docelowej. Do zapisu samego rekordu utworzony zostanie obiekt klasy `KeyValue`, który jako parametry przyjmuje wartość klucza, nazwę rodziny kolumn oraz nazwę kolumny docelowej. Taka para klucz-wartość zostaje przekazana dalej do kontekstu programu. Jako, że nie zdefiniowana została klasa redukująca, powyższe rezultaty są traktowane jako rezultat działania programu MapReduce. Na listingu 4.17. został przedstawiony kod główny programu.

```

public class MaxValueMultipleInput {
    public static void main(String[] args) throws Exception {
        Scan scan = new Scan();
        scan.setCaching(500);
        scan.setCacheBlocks(false);

        Configuration conf = new Configuration();
        conf.set("hbase.table.name", "MaxValuesRegion");

        Job job = new Job();
        job.setJarByClass(MaxValueMultipleInput.class);
        job.setJobName("Zapis do bazy HBase.");

        MultipleInputs.addInputPath(job, new Path(
            "/user/cloudera/input_data/HBase_file.txt"),
            TextInputFormat.class,
            MaxValueMultipleInputFileMapper.class);

        MultipleInputs.addInputPath(job, new Path(
            "/user/cloudera/input_data/HBase_file2.txt"),
            TextInputFormat.class,
            MaxValueMultipleInputFileMapper.class);

        job.setMapOutputKeyClass(ImmutableBytesWritable.class);
        job.setMapOutputValueClass(KeyValue.class);

        job.setOutputFormatClass(HFileOutputFormat.class);

        FileOutputFormat.setOutputPath(job, new Path("output"));
        HTable hTable =
            new HTable(job.getConfiguration(), "MaxValuesRegion");

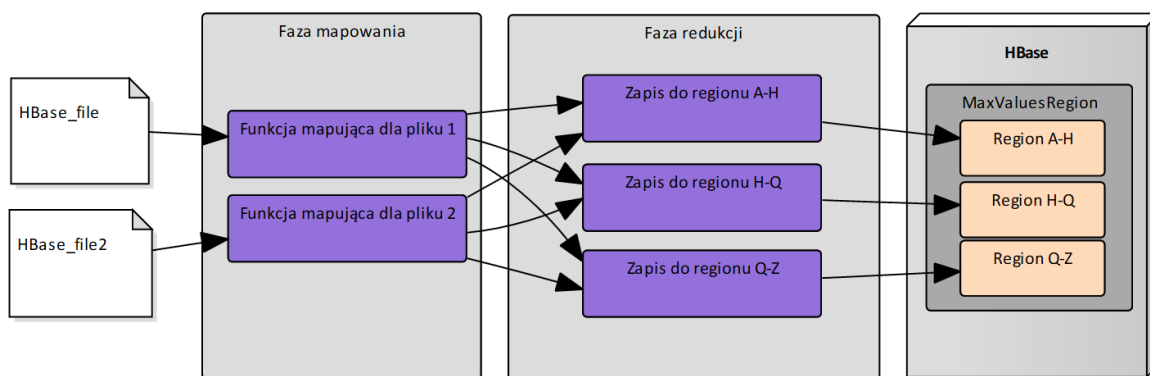
        HFileOutputFormat.configureIncrementalLoad(job, hTable);
        job.waitForCompletion(true);

        if (job.isSuccessful()) {
            HBaseConfiguration.addHbaseResources(job.getConfiguration());
            LoadIncrementalHFiles loadFfiles = new LoadIncrementalHFiles(
                job.getConfiguration());
            loadFfiles.doBulkLoad(new Path("output"), hTable);
        }
    }
}

```

Listing 4.17. Kod głównej klasy programu zapisującego dane z plików do bazy HBase.

Działanie całego programu zostało przedstawione na rysunku 4.7. Dzięki przekazaniu danych konfiguracyjnych odnośnie tabeli docelowej, MapReduce jest w stanie automatycznie wykryć liczbę regionów, partycjonować dane oraz załadować je używając odpowiedniej liczby zadań redukujących.



Rysunek 4.7. Schemat obrazujący działanie programu zapisującego dane do poszczególnych regionów w tabeli HBase.

W tym przykładzie należy zwrócić uwagę na kilka istotnych rzeczy: *setMapOutputKeyClass* oraz *setMapOutputValueClass* konfiguruje typy wyjściowe z kroku mapującego w danym programie. Bardzo istotnym elementem jest tutaj konfiguracja formatu klasy wyjściowej *job.setOutputFormatClass(HFileOutputFormat.class)* która informuje cały model o tym, że rezultaty należy zapisać do plików HFile, są to pliki w których przechowywane są informacje w bazie HBase. Dodatkowo, aby sposób zapisu do bazy był optymalny, należy przekazać odpowiednie informacje o regionach do programu. Za konfigurację oraz stworzenie plików (do odpowiednich regionów) służy metoda *configureIncrementalLoad*, która jako parametr przyjmuje instancje programu oraz instancje klasy *HTable*, która zawiera informacje o tabeli docelowej. Następnie, wewnątrz instrukcji *if* odbywa się już ładowanie z plików bezpośrednio do bazy HBase. Pobierana jest informacja o konfiguracji programu, a także tworzona instancja klasy *LoadIncrementalHFiles*, która posiada metodę *doBulkLoad*. Metoda ta jako parametr pobiera lokalizację folderu, w którym znajdują się pliki *HFile* oraz informację o tabeli docelowej. Gdyby zaistniała potrzeba określenia klasy redukującej dla zapisu do bazy HBase można by było dodać klasę dziedziczącą po klasie *TableReducer* i implementującą metodę *reduce*. Bardzo istotnym elementem jest tutaj odpowiednie ustawienie regionów, gdyż ma to bezpośredni wpływ na ilość zadań redukujących programu. Dla każdego regionu tworzone są oddzielnie zadania redukujące.

### Uruchomienie oraz analiza programu zapisującego dane do bazy HBase

Aby poprawnie uruchomić program należy upewnić się, że pliki z danymi znajdują się w systemie HDFS. Listing 4.18. przedstawia listę operacji jakie należy wykonać aby przygotować dane oraz uruchomić program.

```

hadoop fs -copyFromLocal HBase_file.txt
            /user/cloudera/input_data/HBase_file.txt
hadoop fs -copyFromLocal HBase_file2.txt
            /user/cloudera/input_data/HBase_file2.txt
export HADOOP_CLASSPATH=$(hadoop classpath)
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hbase/lib/*
export HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hbase/*
javac -classpath ${HADOOP_CLASSPATH} *.java
jar -cvf MultipleInputs.jar *.class
hadoop jar MultipleInputs.jar MaxValueMultipleInput

```

Listing 4.18. Przygotowanie danych oraz uruchomienie programu.

Na listingu 4.19. przedstawione zostały dzienniki po uruchomieniu programu. Program MapReduce jest w stanie dokładnie obliczyć liczbę partycji w danej bazie HBase oraz przypisać odpowiednią liczbę operacji redukujących aby zwiększyć wydajność zapisu. Dodatkowo, dane służące do zapisu do poszczególnych partycji są przechowywane w plikach w systemie HDFS i są dostępne dla wszystkich operacji w danym programie. Każda z operacji redukujących może być wykonywana na oddzielnej maszynie, a dokładniej na tej, na której znajduje się konkretny region bazy HBase. Ostatni fragment dzienników zadań informuje o tym jaki plik *HFile*, będzie załadowany do bazy HBase.

```

18/06/02 10:55:26 INFO mapreduce.HFileOutputFormat2: Looking up current regions for table MaxValuesRegion
18/06/02 10:55:26 INFO mapreduce.HFileOutputFormat2: Configuring 3 reduce partitions to match current region count
18/06/02 10:55:26 INFO mapreduce.HFileOutputFormat2: Writing partition information to /user/cloudera/hbase-
staging/partitions_2561596d-f75e-42a7-844c-22be2fd91ece
[...]
18/06/02 10:55:29 INFO input.FileInputFormat: Total input paths to process : 2
18/06/02 10:55:29 INFO mapreduce.JobSubmitter: number of splits:8
[...]
18/06/02 10:55:30 INFO mapreduce.Job: Running job: job_1527955118689_0010
18/06/02 10:55:43 INFO mapreduce.Job: map 0% reduce 0%
18/06/02 10:56:08 INFO mapreduce.Job: map 1% reduce 0%
[...]
18/06/02 12:03:43 INFO mapreduce.LoadIncrementalHFiles: Trying to load
hfile=hdfs://quickstart.cloudera:8020/user/cloudera/output/values/471d7763dddc42fd94eee3bb7b40532d first=HAAAAEEVWHGG
last=PZZZZYRKXLSU
18/06/02 12:03:43 INFO mapreduce.LoadIncrementalHFiles: Trying to load
hfile=hdfs://quickstart.cloudera:8020/user/cloudera/output/values/14dd4fb877eb4e8b82d4832edfc77cd0 first=QAAAAKIDEGWA
last=ZZZZZTFVQANO
18/06/02 12:03:43 INFO mapreduce.LoadIncrementalHFiles: Trying to load
hfile=hdfs://quickstart.cloudera:8020/user/cloudera/output/values/c31aef50879d41c8af6d116daed62daa first=AAAAAEXBUHLV
last=GZZZZYXRMWYB

```

Listing 4.19. Kluczowe wpisy do dziennika zadań informujące o przygotowaniu oraz przebiegu programu zapisującego dane do bazy HBase.

Czasy wykonania programu prezentują się następująco:

- Całkowity czas wykonania programu: 8 minut i 31 sekund
- Średni czas zadania mapującego: 1 minuta i 36 sekund
- Średni czas zadania redukującego: 2 minuty i 39 sekund
- Liczba zadań mapujących jest równa 8, natomiast liczba zadań redukujących wyniosła 3
- Liczba przeprosesowanych plików: 2, łączna wielkość plików: 1 GB danych tekstowych.

## 5. Analiza danych ankietowych

### 5.1. Opis założeń oraz przygotowanie kodu źródłowego

W tym rozdziale zostało ukazane praktyczne zastosowanie implementacji modelu MapReduce do analizy realnych danych statystycznych. Dane zostały zebrane podczas jednej z ankiet przeprowadzonych w Stanach Zjednoczonych. Ankieta zawiera informacje o stanie majątkowym obywateli, np. ilość osób w rodzinie, miejsce zamieszkania, dochód na osobę, wysokość płaconego czynszu czy chociażby dostęp do Internetu. Celem zadania będzie oszacowanie procentu społeczeństwa amerykańskiego posiadającego dostęp do Internetu. Do analizy wykorzystano dwa pliki o wielkości około 600MB każdy oraz mniejszy plik będący słownikiem danych. W ankiecie wzięło udział około trzy miliony osób, a więc w obu plikach znajduje się około trzy miliony rekordów danych. Publicznie dostępne dane zostały pobrane ze strony <https://www.kaggle.com/census/2015-american-community-survey>. Na listingu 5.1. przedstawiono klasę mapującą programu dla pliku z danymi. Metoda *map* przekształca linię z pliku na tablicę elementów typu *String*, następnie poszczególne elementy tablicy przypisywane są do zmiennej *output* tworząc tym samym wyjście z kroku mapującego. Kluczem w tym przypadku jest identyfikator stanu, natomiast wartością jest linia tekstu, która zawiera listę atrybutów oddzieloną przecinkiem. Lista atrybutów zawiera następujące informacje:

1. rodzaj dostępu do Internetu
2. liczba osób zamieszkałych w danym gospodarstwie
3. rodzaj gospodarstwa (prywatne, firma).

```
public class CommSurveyMapper extends
    Mapper<LongWritable, Text, Text, Text> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String[] vals = value.toString().split(",");

        if (vals[1].equalsIgnoreCase("SERIALNO"))
            return;
        String serialId = vals[1];
        String stateId = vals[5];

        String internetAccess = vals[11];
        if(internetAccess.isEmpty())
            internetAccess = "N/A";

        String numberOfPeople = vals[9];
        if(numberOfPeople.isEmpty()){
            numberOfPeople = "N/A";
```

```

    }

    String housingType = vals[10];
    if(housingType.isEmpty())
        housingType="N/A";

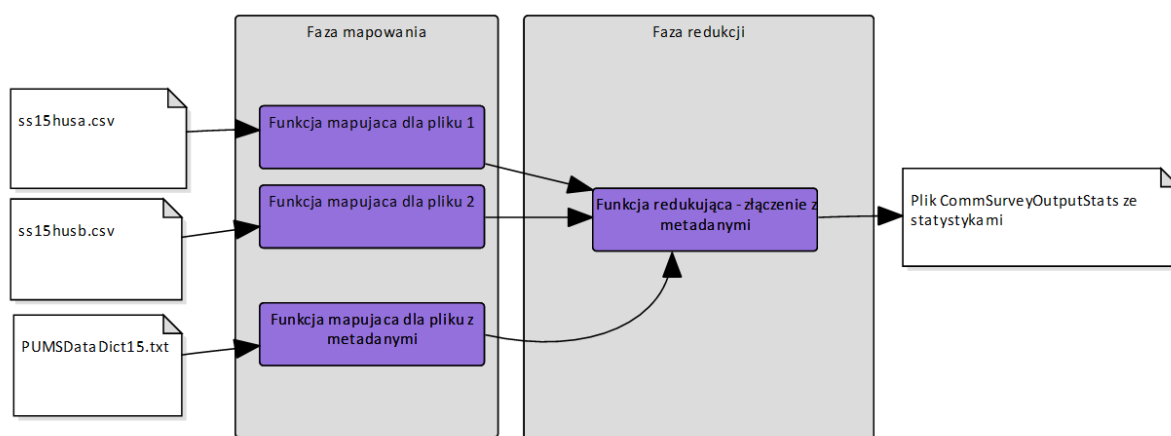
    String output =
        "DATA,"+serialId+","+stateId+","+internetAccess+","+numberOfPeo
        ple+","+","+housingType;

    context.write(new Text(stateId), new Text(output));
}
}

```

Listing 5.1. Klasa mapująca dla pliku z danymi.

Ze względu na to, że plik z danymi nie zawiera informacji o stanie, jedynie jego identyfikator, potrzebne jest również pobranie słownika danych z informacjami o stanach i identyfikatorach. Rysunek 5.1. przedstawia schemat działania programu.



Rysunek 5.1. Schemat działania programu do analizy danych ankietowych.

Kod z listingu 5.2. realizuje zadanie mapujące dla pliku będącego słownikiem danych. Metoda *map* pobiera linię danych oraz tworzy z niej tablicę elementów typu *String*. Następnie przypisuje poszczególne wartości takie jak: identyfikator stanu, nazwa stanu oraz jego skrót. Ważnym elementem tutaj jest dodanie przedrostka *META* do zmiennej *output*, która przekazywana jest na wyjście klasy mapującej. Dzięki temu klasa redukująca będzie w stanie rozpoznać, które dane pochodzą z klasy mapującej odpowiedzialnej za słownik danych, a które za realne dane. Kluczem w tym przypadku jest również identyfikator stanu. Dzięki temu, że obie klasy mapujące zwracają klucz o takiej samej wartości biznesowej, możliwe będzie połączenie danych ze słownikiem danych, czyli wykonania tzw. operacji *JOIN* [9].

```

public class CommSurveyMetadataMapper extends
    Mapper<LongWritable, Text, Text, Text> {

    @Override
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {

        String line = value.toString();
        if(!line.contains("State Code"))
            return;
        else{
            String[] vals = line.split(",");
            String stateId = vals[1];
            String stateName = vals[2];
            String stateShortcut = vals[3];
            String output =
                "META,"+stateId+", "+stateName+", "+stateShortcut;
            context.write(new Text(stateId), new Text(output));
        }
    }
}

```

Listing 5.2. Klasa mapująca dla słownika danych.

Na listingu 5.3. przedstawiony jest kod klasy redukującej. Na początku tworzony jest obiekt klasy *CommSurveyOutputStats*, który będzie odpowiedzialny za przechowywanie rezultatów obliczeń, a także za implementację metod służących do zapisu danych do systemu HDFS. Reduktor w tym przypadku odpowiada za określenie czy dane dla podanego klucza pochodzą z pliku tekstowego czy z słownika danych. W pierwszym przypadku zbierane są dane ankietowe, przypisywane są one do klasy zawierającej rezultaty obliczeń. W drugim przypadku pobierane są informacje o stanie, dla którego przeprowadzane są obliczenia. Klasa redukująca jest w stanie rozróżnić oba typy danych wejściowych dzięki przedrostkowi *DATA* lub *META*.

```

public class CommSurveyReducer extends
    Reducer<Text, Text, Text, CommSurveyOutputStats> {
    @Override
    protected void reduce(Text arg0, Iterable<Text> arg1, Context arg2)
        throws IOException, InterruptedException {
        CommSurveyOutputStats outputStat = new CommSurveyOutputStats();

        for(Text values : arg1){
            String[] vals = values.toString().split(",");
            if(vals[0].equalsIgnoreCase("DATA")){
                String numberOfPeople = vals[4];
                String inType = vals[3];
                Long pplNumber = 0L;
                if(!numberOfPeople.equals("N/A")){
                    pplNumber = Long.parseLong(numberOfPeople);
                }
            }
            else {

```

```

        pplNumber=1L; // assume at least one person
    }
    outputStat.numberOfPeople+=pplNumber;
    outputStat.incrStats(inType,pplNumber);
}
else if(vals[0].equalsIgnoreCase("META")){
    outputStat.stateName = vals[2];
    outputStat.stateShortcut = vals[3];
}
}
outputStat.calculateAverages();
outputStat.stateId = arg0.toString();
arg2.write(arg0, outputStat);
}
}

```

Listing 5.3. Klasa redukująca programu odpowiedzialna za połączenie danych ze słownikami oraz wykonanie obliczeń.

Dane zapisywane są do oddzielnej klasy *CommSurveyOutputStats* implementującej interfejs *Writable* pochodzący z biblioteki głównej Hadoop. Dzięki temu klasa może być bezpośrednio zapisywana jako rezultat obliczeń do kontekstu aplikacji. Możliwe jest również przekazywanie klienckich klas pomiędzy operacjami *Map* oraz *Reduce*, jednakże w tym przypadku jeden obiekt typu *String* działa szybciej, gdyż poszczególne wartości oddzielone są przecinkami i odpowiednio obsługiwane w klasie redukującej. Na listingu 5.4. przedstawiono kod klasy *CommSurveyOutputStats*.

```

public class CommSurveyOutputStats implements Writable {

    public String stateId;
    public String stateName;
    public String stateShortcut;
    public Long numberOfPeople = 0L;
    public Long npWithInternetSubscription = 0L;
    public Long npWithInternetNoSubscription = 0L;
    public Long npWithoutInternet = 0L;
    public Long npDontSpecifyInternetAccess = 0L;
    public String percentpWithInternetSubscription;
    public String percentpWithInternetNoSubscription;
    public String percentpWithoutInternet;
    public String percentpDontSpecifyInternetAccess;

    public void calculateAverages() {
        percentpWithInternetSubscription =
            String.format("%.2f", ((double) npWithInternetSubscription /
                numberOfPeople));
        percentpWithInternetNoSubscription
            =String.format("%.2f", ((double) npWithInternetNoSubscription /
                numberOfPeople));
        percentpWithoutInternet = String.format("%.2f", ((double)
            npWithoutInternet / numberOfPeople));
        percentpDontSpecifyInternetAccess =
            String.format("%.2f", ((double) npDontSpecifyInternetAccess /
                numberOfPeople));
    }
}

```

```

    }

    public void incrStats(String internetType, Long adds) {
        if (internetType.equalsIgnoreCase("N/A"))
            npDontSpecifyInternetAccess = npDontSpecifyInternetAccess
                + adds;
        if (internetType.equalsIgnoreCase("1"))
            npWithInternetSubscription = npWithInternetSubscription +
                adds;
        if (internetType.equalsIgnoreCase("2"))
            npWithInternetNoSubscription =
                npWithInternetNoSubscription + adds;
        if (internetType.equalsIgnoreCase("3"))
            npWithoutInternet = npWithoutInternet + adds;
    }

    @Override
    public void readFields(DataInput arg0) throws IOException {
        stateId = arg0.readUTF();
        stateName = arg0.readUTF();
        stateShortcut = arg0.readUTF();
        numberOfPeople = arg0.readLong();
        npWithInternetSubscription = arg0.readLong();
        npWithInternetNoSubscription = arg0.readLong();
        npWithoutInternet = arg0.readLong();
        npDontSpecifyInternetAccess = arg0.readLong();
        percentpWithInternetSubscription = arg0.readUTF();
        percentpWithInternetNoSubscription = arg0.readUTF();
        percentpWithoutInternet = arg0.readUTF();
        percentpDontSpecifyInternetAccess = arg0.readUTF();
    }

    @Override
    public void write(DataOutput arg0) throws IOException {
        arg0.writeUTF(stateId);
        arg0.writeUTF(stateName);
        arg0.writeUTF(stateShortcut);
        arg0.writeLong(numberOfPeople);
        arg0.writeLong(npWithInternetSubscription);
        arg0.writeLong(npWithInternetNoSubscription);
        arg0.writeLong(npWithoutInternet);
        arg0.writeLong(npDontSpecifyInternetAccess);
        arg0.writeUTF(percentpWithoutInternet);
        arg0.writeUTF(percentpWithInternetNoSubscription);
        arg0.writeUTF(percentpWithoutInternet);
        arg0.writeUTF(percentpDontSpecifyInternetAccess);
    }

    @Override
    public String toString() {
        return "CommSurveyOutputStats [STATE_ID=" + stateId
            + ", STATE_NAME=" + stateName
            + ", STATE_SHORTCUT=" + stateShortcut
            + ", NUMBER_OF_PEOPLE=" + numberOfPeople
            + ", PEOPLE_WITH_INTERNET_SUBS=" +
                npWithInternetSubscription
            + ", PEOPLE_WITH_INTERNET_NO_SUBS="
                + npWithInternetNoSubscription + ",
                PEOPLE_WITHOUT_INTERNET="
                + npWithoutInternet + ",

```

```

        PEOPLE_DONT_SPECIFY_INTERNET_ACCESS="
        + npDontSpecifyInternetAccess
        + ", PRCT_PEOPLE_WITH_INTERNET_SUBS="
        + percentpWithInternetSubscription
        + ", PRCT_PEOPLE_WITH_INTERNET_NO_SUBS="
        + percentpWithInternetNoSubscription
        + ", PRCT_PEOPLE_WITHOUT_INTERNET=" +
        percentpWithoutInternet
        + ", PRCT_PEOPLE_DONT_SPECIFY_INTERNET_ACCESS="
        + percentpDontSpecifyInternetAccess + "];";
    }
}

```

Listing 5.4. Kod klasy przechowującej wyniki obliczeń oraz umożliwiający przekazanie ich do kontekstu programu – na wyjście kroku *Reduce*.

Klasa *CommSurveyOutputStats* zawiera poniższe elementy:

1. Deklaracje pól, które przechowują dane statystyczne dla danego stanu
2. Metodę *incrStats*, która odpowiada za modyfikacje pól
3. Metodę *calculateAverages* odpowiadającą za obliczenie średnich wartości dla danego stanu
4. Implementację metody *readFields* z interfejsu *Writable* służącej do odczytu danych przez kontekst aplikacji
5. Implementację metody *write* służącą do zapisu danych do kontekstu aplikacji
6. Przesłoniętą metodę *toString*, która będzie użyta przy zapisie danych do pliku.

Kod główny programu znajduje się na listingu 5.5. Pierwszym elementem programu jest utworzenie instancji klasy *Job*, która reprezentuje nasz program. Następnie, dodane są przy pomocy metody *addInputPath* klasy *MultipleInputs*, różne pliki wejściowe – z danymi oraz słownik danych. Dla każdego rodzaju plików dodana jest odpowiednia klasa mapująca. Na końcu dodane są informacje o typach wyjściowych z poszczególnych kroków *Map* oraz *Reduce*.

```

public class CommSurveyMain {

    public static void main(String[] args) throws IOException,
        ClassNotFoundException, InterruptedException {
        Job job = new Job();
        job.setJarByClass(CommSurveyMain.class);
        job.setJobName("Analiza statystyczna.");
        MultipleInputs.addInputPath(job, new Path(
            "/home/cloudera/Desktop/untitled folder/2015-american-
            community-survey/ss15husa.csv"),
            TextInputFormat.class, CommSurveyMapper.class);
        MultipleInputs.addInputPath(job, new Path(
            "/home/cloudera/Desktop/untitled folder/2015-american-
            community-survey/ss15husb.csv"),
            TextInputFormat.class, CommSurveyMapper.class);
    }
}

```

```

        MultipleInputs.addInputPath(job, new
        Path("PUMSDataDict15.txt"),
        TextInputFormat.class, CommSurveyMetadataMapper.class);
        FileOutputFormat.setOutputPath(job, new Path("output"));
        job.setReducerClass(CommSurveyReducer.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(CommSurveyOutputStats.class);
        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}

```

Listing 5.5. Klasa główna programu analizującego dane statystyczne przy użyciu MapReduce.

## 5.2. Przedstawienie rezultatów działania programu

Przed uruchomieniem programu należy umieścić wszystkie pliki w systemie HDFS, np. uruchamiając polecenia z listingu 5.6.

```

hadoop fs -mkdir /user/cloudera/home/2015-american-community-survey

hadoop fs -copyFromLocal ss15husa.csv /user/cloudera/home/2015-american-
community-survey/ss15husa.csv
hadoop fs -copyFromLocal ss15husb.csv /user/cloudera/home/2015-american-
community-survey/ss15husb.csv
hadoop fs -copyFromLocal PUMSDataDict15.txt /user/cloudera/home/2015-
american-community-survey/PUMSDataDict15.txt

export HADOOP_CLASSPATH=$(hadoop classpath)
javac -classpath ${HADOOP_CLASSPATH} *
jar -cvf CommSurvey.jar *.class

hadoop jar CommSurvey.jar CommSurveyMain

```

Listing 5.6. Przygotowanie oraz uruchomienie programu do analizy danych statystycznych.

Wynikiem wykonania powyższych poleceń jest uruchomienie programu MapReduce. Z dzienników zadania dostępnych pod wskazanym adresem URL po wykonaniu programu można odczytać następujące informacje:

- Czas trwania całego programu: 1 minuta 32 sekundy
- Średni czas trwania kroku mapującego to 20 sekund
- Średni czas trwania kroku redukującego to 5 sekund
- Liczba operacji mapujących wyniosła 11 (po 5 na każdy plik oraz 1 na plik z metadanymi) oraz 1 operacja redukująca

Dodatkowo, program rozpoczął kroki redukujące jeszcze zanim zakończono kroki mapujące, pozwala to skrócić czas trwania całego programu. Innym istotnym elementem jest

fakt, że program MapReduce potrzebuje sporo czasu na komunikację z planistą zasobów, zarządcami węzłów oraz na dostęp do zasobów w systemie plików HDFS. Z tego powodu MapReduce nadaje się świetnie do przetwarzania plików wsadowych, kiedy rezultaty nie muszą być dostępne natychmiast. MapReduce słabo sprawdza się natomiast kiedy potrzebny jest ciągły dostęp do najświeższych danych. Rezultatem wykonania programu jest plik part-r-00000 z zebranymi informacjami o liczbie osób posiadających dostęp do Internetu dla danego stanu. Fragment pliku wyjściowego z rezultatami analizy pokazano na listingu 5.7.

```
06  CommSurveyOutputStats [STATE_ID=06,
    STATE_NAME=California,
    STATE_SHORTCUT=CA,
    NUMBER_OF_PEOPLE=374943,
    PEOPLE_WITH_INTERNET_SUBS=310032,
    PEOPLE_WITH_INTERNET_NO_SUBS=10930,
    PEOPLE_WITHOUT_INTERNET=38640,
    PEOPLE_DONT_SPECIFY_INTERNET_ACCESS=15341,
    PRCT_PEOPLE_WITH_INTERNET_SUBS=0.83,
    PRCT_PEOPLE_WITH_INTERNET_NO_SUBS=0.03,
    PRCT_PEOPLE_WITHOUT_INTERNET=0.10,
    PRCT_PEOPLE_DONT_SPECIFY_INTERNET_ACCESS=0.04]

08  CommSurveyOutputStats [STATE_ID=08,
    STATE_NAME=Colorado,
    STATE_SHORTCUT=CO,
    NUMBER_OF_PEOPLE=53570,
    PEOPLE_WITH_INTERNET_SUBS=45298,
    PEOPLE_WITH_INTERNET_NO_SUBS=1500,
    PEOPLE_WITHOUT_INTERNET=4676,
    PEOPLE_DONT_SPECIFY_INTERNET_ACCESS=2096,
    PRCT_PEOPLE_WITH_INTERNET_SUBS=0.85,
    PRCT_PEOPLE_WITH_INTERNET_NO_SUBS=0.03,
    PRCT_PEOPLE_WITHOUT_INTERNET=0.09,
    PRCT_PEOPLE_DONT_SPECIFY_INTERNET_ACCESS=0.04]
```

Listing 5.7. Rezultaty obliczeń – dane statystyczne mieszkańców USA odnośnie dostępu do Internetu.

Na podstawie listingu widać, że dla dwóch stanów około 85% ludzi ma stały dostęp do Internetu, około 10% ludzi deklaruje, że nie ma dostępu do Internetu. Świadczą o tym wartości zmiennej `PRCT_PEOPLE_WITH_INTERNET_SUBS`, która informuje o tym jaki procent osób posiada dostęp do Internetu w danym stanie oraz `PRCT_PEOPLE_WITHOUT_INTERNET`, która informuje jaki procent osób takiego dostępu nie posiada.

### 5.3. Wykorzystanie MapReduce jako API wyższego poziomu

Analizując dokładnie otrzymane rezultatu oraz działanie programu można zauważyć, że wykonane zostało zwykle zapytanie SQL na plikach w klastrze. Wynik rezultatów całego powyższego programu można zobrazować przy użyciu zapytania SQL ukazanego na listingu 5.8. Zapytanie SQL nie pochodzi z żadnego konkretnego dialektu a jedynie obrazuje ideę programu.

```
SELECT data.STATE_ID,
metadata.STATE_NAME,
metadata.STATE_SHORTCUT,
COUNT(*) NUMBER_OF_PEOPLE,
SUM(IF(InternetType="1")) AS PEOPLE_WITH_INTERNET_SUBS,
SUM(IF(InternetType="2")) AS PEOPLE_WITH_INTERNET_NO_SUBS,
SUM(IF(InternetType="3")) AS PEOPLE_WITHOUT_INTERNET,
SUM(IF(InternetType="N/A")) AS PEOPLE_DONT_SPECIFY_INTERNET_ACCESS,
SUM(IF(InternetType="1"))/COUNT(*) AS PRCT_PEOPLE_WITH_INTERNET_SUBS,
SUM(IF(InternetType="2"))/COUNT(*) AS PRCT_PEOPLE_WITH_INTERNET_NO_SUBS,
SUM(IF(InternetType="3"))/COUNT(*) AS PRCT_PEOPLE_WITHOUT_INTERNET,
SUM(IF(InternetType="N/A"))/COUNT(*) AS
PRCT_PEOPLE_DONT_SPECIFY_INTERNET_ACCESS
FROM (SELECT * FROM "ss15husa.csv" UNION ALL SELECT * FROM "ss15husb.csv")
data
INNER JOIN "PUMSDDataDict15.txt" metadata ON (data.STATE_ID =
metadata.STATE_ID)
GROUP BY data.STATE_ID, metadata.STATE_NAME, metadata.STATE_SHORTCUT;
```

Listing 5.8. Logiczna reprezentacja wykonanego zadania przy użyciu składni języka SQL.

Powyższy kod jest dużo czytelniejszy i zdecydowanie prostszy do napisania oraz analizy niż programy MapReduce pisane w Javie. W klastrze dostępne są już odpowiednie narzędzia, które potrafią tłumaczyć język SQL na programy MapReduce i analizować pliki, które przybierają formy tabelaryczne tak jak w przypadku zwykłych baz relacyjnych. Takie rozwiązania udostępnia już między innymi Apache Hive. Hive pozwala na utworzenie tabel na istniejących plikach tekstowych oraz ich analizę przy użyciu zapytań języka HQL *Hive Query Language*, który oparty jest na języku SQL. Wewnętrznie, zapytania HQL są przetwarzane na programy MapReduce oraz uruchamiane w klastrze do pobrania odpowiednich danych [10].

### 5.4. Dyskusja

Podsumowanie wszystkich analiz zostało przedstawione w tabeli 5.1. Na podstawie takiego zestawienia oraz wniosków znajdujących się po każdej analizie można stwierdzić kilka istotnych faktów odnośnie programowania przy użyciu MapReduce.

Podrozdział pracy	Rodzaj przetwarzania	Dane wejściowe	Dane wyjściowe	Dostępne zasoby	Całkowity czas przetwarzania
4.1	Przykładowy program MapReduce	420MB danych, 4 pliki jednakowej wielkości	Plik zawierający maksymalną wartość dla danego identyfikatora	4GB RAM, 2vCPU	8 min 21 sekund
4.2	Przykładowy program MapReduce – różne dane wejściowe w tym odczytanie wartości z bazy/ partycjonowanie	10GB danych	Plik zawierający maksymalną wartość dla danego identyfikatora	13,4GB RAM, 4vCPU	13 min 14 sekund
4.3	Program MapReduce czytający dane z pliku i zapisujący do bazy NoSQL	1GB danych	Dane dostępne w bazie NoSQL	13,4GB RAM, 4vCPU	8 minut 31 sekund
5	Analiza danych statystycznych	1GB danych tekstowych + metadane	Plik zawierający rezultaty obliczeń	13,4GB RAM, 2vCPU	1 minuta 32 sekundy

Tabela 5.1. Zestawienie wszystkich wykonanych analiz.

Zwiększenie szybkości i wydajności działania programów można osiągnąć poprzez dodanie dodatkowych zasobów na jednej z maszyn lub też dodanie zupełnie nowych węzłów roboczych do klastra komputerowego. Ze względu na to, że MapReduce korzysta z systemu YARN nowe zasoby zostaną wykryte oraz użyte automatycznie, nie potrzeba konfigurować żadnych dodatkowych elementów ani zmieniać definicji programu. Dzięki temu skalowalność systemu przebiega w prosty sposób oraz ma charakter liniowy. Im więcej kontenerów jest dostępnych w klastrze komputerowym tym szybciej działać będzie program. Dzięki dostępności rozwiązań chmurowych możliwości skalowania systemów są bardzo duże, w dużych systemach produkcyjnych w kilka godzin można zwiększyć wydajność procesów o kilkaset lub nawet kilka tysięcy procent poprzez dodanie nowych zasobów lub całych węzłów roboczych. Niestety wiąże się to z reguły z dużymi kosztami. Dzięki temu, że MapReduce komunikuje się bezpośrednio z systemem YARN, pozostaje on jednak jednym z najefektywniejszych modeli działających na platformie Hadoop.

Wykonanie programu nawet na małych zbiorach danych potrafi zająć od kilku do kilkunastu minut. MapReduce świetnie sobie radzi z analizowaniem dużych zbiorów danych, ale wykonywanie prostych operacji zajmuje zdecydowanie zbyt dużo czasu. Wynika to z faktu, że program musi sprawdzić dane, inicjalizować procesy czy też negocjować zasoby. Z tego powodu programy pracujące w tym modelu nie powinny być używane bezpośrednio do wykonywania analiz czasu rzeczywistego. Przykładem jest tutaj ostatnia analiza dotycząca

korzystania z Internetu przez społeczeństwo amerykańskie, do przetworzenia niewielkiej ilości danych program potrzebował aż 90 sekund. W przypadku kiedy użytkownik końcowy potrzebuje wygenerować raport na małej ilości danych, czas trwania programu wynoszący 90 sekund może być zbyt długim czasem oczekiwania i powodować zmniejszenie efektywności pracy oraz liczby analiz dokonywanych przez użytkowników w ciągu dnia.

Innym mankamentem modelu jest również brak dostępności do operacji pośrednich. Użytkownik ma dostęp jedynie do danych źródłowych oraz końcowych, nie ma możliwości podejrzenia danych w trakcie wykonywania programu. Programy MapReduce z reguły zawierają wiele przekształceń – nie ma wtedy możliwości obejrzenia rezultatów po kilku wybranych operacjach. Chcąc podglądać rezultaty pośrednich obliczeń należy dodatkowo zapisywać je na dysk lub do innego systemu plików. Przykładem narzędzia, które oferuje taką możliwość jest Spark. Dzięki interaktywnemu trybowi pracy Spark umożliwia przeglądanie pracy po niemal każdej transformacji bez dodatkowych operacji wpływających na cały program.

O ile analizy czasu rzeczywistego nie są najlepszym zastosowaniem dla MapReduce, o tyle jego użycie do przetwarzania dużych plików wsadowych w statyczny sposób jest idealnym rozwiązaniem. Typowym zastosowaniem modelu jest scenariusz, w którym istnieje potrzeba pobrania danych z różnych źródeł, np. z dzienników pochodzących ze stron internetowych, przekształcenia do odpowiedniej formy, integracja oraz przekazanie ich do innego narzędzia, które umożliwia szybsze analizowanie ustrukturyzowanych danych. Przykładem może być tutaj analiza z rozdziału 4.2. W przeprowadzonej analizie zintegrowano dane pochodzące z plików tekstowych i bazy NoSQL, ale nie ma żadnych przeciwwskazań, aby używać innych źródeł np. baz relacyjnych. Innym zastosowaniem MapReduce są analizy statyczne. Przykładem może być tutaj proces, który pod koniec dnia pobierze oraz policzy ilość wystąpień danych słów kluczowych wpisanych na witrynach internetowych danego sklepu.

Na podstawie analizy z rozdziału piątego można zauważyć, że udało się zrealizować zadanie stworzenia rozproszonego zapytania SQL na plikach znajdujących się w systemie HDFS. Wymagało to jednak dość sporo pracy i pisanie zaawansowanego kodu w języku Java. Pisanie procesów MapReduce jest czasochłonne i skomplikowane, natomiast umożliwia niskopoziomową manipulację danych, nawet na poziomie bajtów. Użytkownicy biznesowi chcący dokonać analizy części danych w klastrze musieliby definiować klasy mapujące i redukujące dla każdej pojedynczej analizy co szybko mogłoby doprowadzić do ogromnych kosztów, nieproporcjonalnych do uzyskanej wiedzy z poszczególnych analiz. MapReduce

w swoim działaniu jest jednak dość generyczny a dobrze zorganizowane API, które udostępnia jest wykorzystywane przez narzędzia wyższego poziomu, np. Hive.

Wszystkie procesy wykonywane w klastrze komputerowym można podzielić na dwa rodzaje: analizy wykonywane przez użytkowników na danych znajdujących się w klastrze oraz procesy służące do ekstrakcji, integracji oraz manipulacji danych. MapReduce zdecydowanie bardziej nadaje się do drugiego rodzaju procesów, głównie do przetwarzania danych dużych rozmiarów na niskim poziomie abstrakcji lub tworzeniu raportów statycznych. Do przeglądania danych lub tworzenia raportów dynamicznych lepiej jest używać innych narzędzi, np. Hive lub Spark.

## 6. Podsumowanie

W pracy dokonany został przegląd możliwości modelu MapReduce do analizy danych. W części teoretycznej przedstawione zostały najważniejsze elementy platformy Hadoop takie jak: system YARN, rozproszony system plików HDFS oraz baza HBase. Przedstawienie tych narzędzi miało za zadanie pomóc w zrozumieniu programów zaprezentowanych w kolejnych rozdziałach pracy. W tej części omówiono także model MapReduce, sposób działania programów w klastrze komputerowym oraz instrukcje do instalacji i weryfikacji środowiska uruchomieniowego. W części praktycznej wykonano analizy obrazujące możliwości implementacji modelu na platformie Hadoop. Zaprezentowano programy, które pokazują działanie podstawowych funkcjonalności MapReduce, z których korzystać może programista podczas pisania aplikacji. Programy zawierały przykłady implementacji kroków głównych *Map* oraz *Reduce*, kroków opcjonalnych takich jak *Combining* oraz *Partitioning*, a także przykłady możliwości odczytu i zapisu danych pochodzących z różnych źródeł: plików tekstowych oraz bazy danych. W rozdziale piątym wykonana została analiza danych ankietowych dotyczących m.in. posiadania połączenia Internetowego w Stanach Zjednoczonych. Analiza została przeprowadzona na kilku plikach tekstowych w tym słownikach danych, które wczytane były przy pomocy różnych klas mapujących oraz zagregowane i połączone przy pomocy klasy redukującej. W wyniku tej analizy udało się ustalić procent społeczeństwa amerykańskiego posiadającego dostęp do Internetu w poszczególnych stanach.

Autor za własny wkład pracy uważa:

- Dokonanie przeglądu serwisów i narzędzi platformy Hadoop oraz opisanie ich najważniejszych elementów
- Dokonanie przeglądu literatury dotyczącej algorytmu MapReduce oraz przedstawienie w pracy jego głównych założeń
- Opracowanie możliwości implementacji programów w modelu MapReduce
- Napisanie kodu źródłowego i uruchomienie programów w języku Java, skryptów powłoki systemu Linux w języku BASH oraz utworzenie tabel w bazie HBase
- Zebranie i przedyskutowanie uzyskanych wyników.

## Literatura

- [1] Jonathan Seidman, Gwen Shapira, Ted Malaska, Mark Grover: Hadoop Application Architectures, O'Reilly 2015
- [2] Tom White: Hadoop. Kompletny przewodnik, Helion 2015
- [3] Arun C. Murthy, Vinod Kumar Vavilapalli: Apache Hadoop YARN. Moving beyond MapReduce and Batch Processing with Apache Hadoop 2, 2014
- [4] Eric Sammer: Hadoop operations, O'Reilly 2012
- [5] Chuck Lam: Hadoop in action, 2010
- [6] Apache HBase Team: Apache HBase Reference Guide, Version 3.0.0
- [7] Srinath Perera, Thilina Gunarathne: Hadoop MapReduce Cookbook, 2013
- [8] Khaled Tannir: Optimizing Hadoop for MapReduce, PACKT 2014
- [9] Donald Miner, Adam Shook: MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems, O'Reilly 2012
- [10] Jason Rutberglen, Dean Wampler, Edward Capriolo: Programming Hive. Data Warehouse and Query Language for Hadoop, O'Reilly 2012

## Zawartość płyty CD

Do pracy została dołączona płyta CD, która zawiera:

- IŻEWSKI\_Jakub\_EF-ZI\_143917.pdf – treść pracy dyplomowej zapisana w formacie .pdf
- Implementacja.txt – plik tekstowy zawierające informacje odnośnie instalacji systemu uruchomieniowego
- Pliki źródłowe do rozdziału 4.1 – folder zawierający kod programu do analizy plików tekstowych wykonanych w rozdziale 4.1
- Pliki źródłowe do rozdziału 4.2 – folder zawierający kod programu do analizy danych pochodzących z różnych formatów źródłowych dokonanej w rozdziale 4.2
- Pliki źródłowe do rozdziału 4.3 – folder zawierający kod programu do zapisu danych do bazy HBase z rozdziału 4.3
- Pliki źródłowe do rozdziału 5 – folder zawierający kod programu oraz dane słownikowe potrzebne do przeprowadzenia analizy danych ankietowych z rozdziału 5.

Sygnatura:

POLITECHNIKA RZESZOWSKA im. I. Łukasiewicza  
Wydział Elektrotechniki i Informatyki  
Katedra Informatyki i Automatyki

Rzeszów, 2018

### **STRESZCZENIE PRACY DYPLOMOWEJ INŻYNIERSKIEJ**

#### **PRAKTYCZNA WERYFIKACJA MOŻLIWOŚCI MODELU MAPREDUCE DO ANALIZY DANYCH**

Autor: Jakub Iżewski, nr albumu: EFZI - 143917

Opiekun: dr inż. Krzysztof Świder

Słowa kluczowe: MapReduce, Big Data, Hadoop, Java, analiza danych

Celem pracy była analiza modelu MapReduce oraz jego implementacja na platformie Hadoop. Oprócz samego modelu, w pracy zostały opisane główne komponenty platformy Hadoop takie jak: HDFS, YARN oraz HBase. Przygotowanych zostało kilka analiz obrazujących najważniejsze aspekty programowania MapReduce w języku Java: odczyt oraz zapis różnych formatów danych, przykład użycia modelu do analizy danych ankietowych. Na podstawie przeprowadzonych analiz przedstawione zostały mocne i słabe strony modelu oraz przykłady zastosowań w systemach analitycznych.

RZESZOW UNIVERSITY OF TECHNOLOGY  
Faculty of Electrical and Computer Engineering  
Department of Control and Computer Engineering

Rzeszow, 2018

### **DIPLOMA THESIS BS ABSTRACT**

#### **PRACTICAL VERIFICATION OF MAPREDUCE MODEL USAGES IN DATA ANALYSIS**

Author: Jakub Iżewski, code: EFZI - 143917

Supervisor: dr inż. Krzysztof Świder

Key words: MapReduce, Big Data, Hadoop, Java, data analysis

The main goal of diploma thesis was to verify MapReduce model and its implementation on Hadoop platform. The thesis contains technical description of Hadoop main components: YARN, HDFS and HBase. There were few analyses prepared showing most important aspects of programming MapReduce in Java: read and write of different data formats, example of MapReduce usage in survey data analysis. Based on all analyses, there were described most important advantages and disadvantages of the model and presented few examples of MapReduce usage in analytical systems.