Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
WYDZIAŁ FIZYKI I INFORMATYKI STOSOWANEJ

Praca magisterska

Porównanie narzędzi do optymalizacji przetwarzania dużych zbiorów danych

Autor: *Alicja Zoń*
Kierunek studiów: *Informatyka Stosowana*
Specjaloność: *Grafika komputerowa i przetwarzanie obrazów*
Opiekun pracy: *dr inż. Elżbieta Strzałka*

Kraków, 2020

Uprzedzona o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzona o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».", oświadczam, że niniejszą pracę dyplomową wykonałam osobiście i samodzielnie i że nie korzystałam ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. — Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

Kraków, lipiec 2020

Tematyka pracy magisterskiej i praktyki dyplomowej Alicji Zoń, studentki drugiego roku studiów drugiego stopnia na kierunku informatyka stosowana, specjalności grafika komputerowa i przetwarzanie danych

Temat pracy magisterskiej:

Porównanie narzędzi do optymalizacji przetwarzania dużych zbiorów danych

Opiekun pracy:	dr inż. Elżbieta Strzałka
Recenzenci pracy:	
Miejsce praktyki dyplomowej:	WFiIS AGH, Kraków

Program pracy magisterskiej i praktyki dyplomowej

1. Omówienie realizacji pracy magisterskiej z opiekunem.
2. Zebranie i opracowanie literatury dotyczącej tematu pracy.
3. Praktyka dyplomowa:
   - zapoznanie się z wcześniej przygotowaną literaturą,
   - zapoznanie się oraz pogłębienie wiedzy dotyczącej technologii wykorzystywanych w pracy magisterskiej,
   - zebranie i obróbka danych potrzebnych do realizacji tematu pracy,
   - sporządzenie planu działań,
   - sporządzenie sprawozdania z praktyki.
4. Implementacja części programistycznej pracy.
5. Zebranie i analiza otrzymanych wyników.
6. Opracowanie redakcyjne pracy.

Termin oddania w dziekanacie:

....

(podpis kierownika katedry) (podpis opiekuna)

Ocena merytoryczna opiekuna

Ocena merytoryczna recenzenta

Spis treści

1. Wstęp 7
- 1.1. Wprowadzenie 7
- 1.2. Zakres i cel pracy 8
- 1.3. Zawartość pracy 9

2. Rozwiązania do przetwarzania dużych ilości danych 11
- 2.1. Architektura big data 11
  - 2.1.1. Architektura lambda 13
  - 2.1.2. Architektura kappa 13
- 2.2. MapReduce 14
  - 2.2.1. Hadoop MapReduce 15
- 2.3. RDD 16
  - 2.3.1. Apache Spark 16
- 2.4. Zmierzch technologii Hadoop 17

3. Optymalizacja 19
- 3.1. Metodologia pomiarów 19

4. Wykorzystane technologie 21
- 4.1. Ekosystem Hadoop 21
  - 4.1.1. HBase 21
  - 4.1.2. Hive 22
- 4.2. Spark 22
- 4.3. Cassandra 22
- 4.4. Microsoft Azure 22
  - 4.4.1. Databricks 23

5. Przykład obliczeniowy 25
- 5.1. Scala a Java 25
- 5.2. Hadoop i Spark lokalnie 27
- 5.3. Hadoop i Spark w chmurze 28

6. Architektura lokalna ... 31
6.1. Przetwarzanie danych nieustrukturyzowanych ... 31
6.1.1. Klaster Apache Hadoop ... 31
6.1.2. Klaster Apache Spark ... 34
6.1.3. Bazy danych ... 37
6.2. Przetwarzanie danych uporządkowanych ... 38
6.2.1. Spark DataFrame ... 39
6.2.2. Hadoop MapReduce ... 42
6.2.3. Odczyt danych ... 44
6.3. Przetwarzanie danych w czasie rzeczywistym ... 45
6.4. Podsumowanie ... 47
7. Architektura w chmurze Azure ... 49
7.1. Przetwarzanie danych ... 49
7.1.1. HDInsight ... 49
7.1.2. Databricks ... 52
7.2. Wykorzystanie zasobów ... 54
7.3. Skalowalność ... 59
7.4. Podsumowanie ... 61
8. Wnioski ... 63
Bibliografia ... 65

1. Wstęp

1.1 Wprowadzenie

Wraz z postępem technologicznym i cyfryzacją kolejnych aspektów życia, generowane są coraz większe ilości danych. We współczesnym świecie dużą wagę przywiązuje się do wartości, które można z nich wyciągnąć. Tradycyjne systemy nie są jednak w stanie wydajnie i szybko przetworzyć i analizować dużych ilości danych – konieczne jest zastosowanie specjalnych technologii i narzędzi do tego przeznaczonych. Do tego dochodzi również dynamiczny rozwój tak zwanego Internetu Rzeczy [1] (ang. *Internet of Things*), w którym zawierają się między innymi wszelkie przenośne urządzenia do monitorowania zdrowia, mierniki energii czy zanieczyszczeń w miastach, urządzenia do monitorowania zagrożeń, obsługa inteligentnych domów, itd. – wszystko to opiera się na przetwarzaniu i optymalizowaniu dużych ilości danych.

Duże zbiory różnorodnych danych, z których przetwarzania można wyciągnąć nową wiedzę i wartość określa się terminem „big data" – nazwa ta stosowana jest również w języku polskim. Termin ten jest używany już od 1990 roku [2], choć dopiero po roku 2000 zyskał nową dynamikę, wraz z rozwojem sieci internetowej i wyszukiwarek internetowych. Obecnie tym terminem określa się dane, które charakteryzują się następującymi cechami [3] (nazywanymi „zasadą 4-V"):

- **Wielkość danych** (ang. *volume*): określa dane o rozmiarach tera-, petabajtów lub nawet większych. Tak duże ilości danych wymagają specjalnego podejścia do ich przechowywania i przetwarzania.

- **Prędkość przetwarzania** (ang. *velocity*): odnosi się do prędkości, w jakiej dane są generowane – często są to miliony wiadomości i zdarzeń w ciągu sekundy, które wymagają natychmiastowego przetworzenia, aby sprostać wymaganiom przedsiębiorstw i użytkowników.

- **Różnorodność danych** (ang. *variety*): odnosi się do zróżnicowania źródeł, z których dane pochodzą, jak i do różnych typów danych, które mogą być zapisane w różnych formatach, bazach danych, a także do ich postaci: ustrukturyzowanej lub bez określonej struktury. Dane mogą mieć formę tekstową, ale także obrazową, dźwiękową lub wideo. Konieczne jest opracowanie podejścia, które jest w stanie łatwo dostosować się do takiej różnorodności, nie tracąc przy tym na szybkości działania.

- **Jakość danych** (ang. *veracity*): określa wartość, jaką może wnieść analiza zbioru danych. Dane o wysokiej jakości przyczyniają się do wartościowych raportów, z których można wyciągnąć znaczącą analizę i uzyskać nową wiedzę. Zbiór danych może się również składać z elementów wartościowych i tych bez znaczenia, które są pomijane na etapie przetwarzania.

Projektując system big data, nastawiamy się na wielokrotne przetwarzanie zróżnicowanych danych. Podstawowymi wyzwaniami [4], jakie można napotkać, są:

- **Prywatność i bezpieczeństwo** – duże zbiory danych zawierają często wiele personalnych i wrażliwych informacji, na przykład analizując dane dla służby zdrowia, przetwarzane są informacje o pacjentach, ich chorobach i lekach. Konieczne jest więc zapewnienie bezpieczeństwa w przechowywaniu i przetwarzaniu danych.

- **Wyzwania analityczne** – na podstawie przetwarzanych danych są generowane raporty, które prowadzą do podejmowania określonych decyzji biznesowych. Duże znaczenie ma więc dbanie o jakość danych i ich przetwarzania na każdym etapie procesu, a także określenie, jak wielki zbiór danych jest potrzebny, by osiągnąć oczekiwane wyniki.

- **Skalowalność** – system musi być dostosowany do uruchamiania wielu zadań jednocześnie, a także do zwiększającej się ilości danych. Musi to być uwzględnione zarówno na etapie przetwarzania, jak i przechowywania danych.

- **Optymalizacja** – jest to szukanie sposobu na to, by osiągnąć jak najwyższą wydajność przy możliwie najniższych kosztach, biorąc pod uwagę wszelkie ograniczenia i warunki wykonywanego zadania. Celem jest jak najlepsze wykorzystanie dostępnych zasobów, bez konieczności dokładania nowych.

O ile o bezpieczeństwie danych i jakości otrzymywanych wyników myśli się już podczas tworzenia nowego systemu, to skalowalność i optymalizację łatwo pominąć, gdy zbiór danych nie jest jeszcze zbyt duży. Jednak zadanie, które dla 50 MB danych wykona się o 2 sekundy za długo, dla 50 TB może się wykonywać o 2 godziny za długo, więc ważne jest, by szukać możliwości ulepszenia działania już na początku implementacji.

1.2 Zakres i cel pracy

Celem pracy jest przedstawienie i porównanie działania technologii związanych z przetwarzaniem big data. Analizując wady i zalety poszczególnych rozwiązań, brane są też pod uwagę możliwości optymalizacyjne – często domyślna konfiguracja lub pierwsza implementacja programu nie jest tą idealną, więc sprawdzono sposoby na ich ulepszenie. Praca skupia się w głównej mierze na przetwarzaniu danych, ale przedstawiono również zagadnienia związane z architekturą

big data i przechowywaniem danych. Zaimplementowane i analizowane programy pełnią rolę testów wydajnościowych, na podstawie których wyciągnięte zostały poszczególne wnioski. Cały zaimplementowany kod dostępny jest w repozytorium GitHub pod adresem:

`https://github.com/alicjazon/big-data-benchmark`

Praca jest pewnego rodzaju wprowadzeniem do tematyki przetwarzania danych, porównuje działanie poszczególnych technologii, możliwości optymalizacyjne, a także łatwość użycia. Duży nacisk położono na zapewnienie tych samych warunków do porównania, a także różnorodność danych i przykładów, by ostateczne wnioski były miarodajne i przynosiły wartość zarówno dla czytelnika dopiero poznającego opisywane technologie, jak i tego bardziej zaawansowanego, chcącego rozszerzyć swoją wiedzę. Praca ma na celu również przybliżyć różnego rodzaju problemy, które można napotkać podczas przetwarzania danych, oraz zaproponować ich rozwiązania.

1.3 Zawartość pracy

Praca podzielona jest na dwie główne części – teoretyczną i praktyczną. W skład części teoretycznej wchodzą następujące rozdziały:

- **Rozdział 2**: zawiera opis typowej architektury do przetwarzania dużej ilości danych, a także wprowadzenie do analizowanych w tej pracy technologii, wraz z opisem modeli programistycznych, na których oparte jest ich działanie.
- **Rozdział 3**: przedstawia miary wydajności procesu przetwarzania danych, a także szczegóły na temat metodologii pomiarów wykonywanych w niniejszej pracy.
- **Rozdział 4**: zawiera opis wykorzystanych technologii i ich możliwości.

Część praktyczna również podzielona jest na trzy rozdziały:

- **Rozdział 5**: zawiera analizę prostego przykładu obliczeniowego, będącą wprowadzeniem do poszczególnych narzędzi do przetwarzania danych. Zarysowane są w nim główne różnice między technologiami, a także językami programowania.
- **Rozdział 6**: przedstawia działanie poszczególnych narzędzi w oparciu o różne przykłady przetwarzania danych. Cała architektura jest stworzona w oparciu o jeden lokalny komputer, co pozwala na lepszy wgląd w konfigurację i działanie poszczególnych technologii.
- **Rozdział 7**: zawiera analizę działania programów uruchomionych w chmurze obliczeniowej. Działanie poszczególnych klastrów jest porównywane pod kątem szybkości przetwarzania danych, optymalnego wykorzystania zasobów i skalowalności.

Ostatnim rozdziałem jest rozdział 8, który zawiera wnioski i podsumowanie wykonanej analizy.

2. Rozwiązania do przetwarzania dużych ilości danych

2.1 Architektura big data

Ciągła analiza tera- lub petabajtów danych nie jest możliwa do wykonania w jednym zadaniu obliczeniowym, na jednej maszynie. Aby sprostać tym wymaganiom, konieczne było opracowanie odpowiedniej architektury. Musiała ona zawierać mechanizmy do przyjmowania, ochrony, przetwarzania i transformacji danych, a następnie ich zapisu do bazy danych bądź systemu plików. Poszczególne rozwiązania różnią się od siebie, ale ogólnie architekturę big data można podzielić na cztery warstwy logiczne [5]:

- **Źródła danych**: dane mogą pochodzić zarówno z aplikacji i wewnętrznych serwerów organizacji, jak i od zewnętrznych dostawców. Potrzebne jest co najmniej jedno źródło danych, ale często bywa ich więcej – relacyjne bazy danych, hurtownie danych, urządzenia mobilne, media społecznościowe, pliki dzienników serwerów internetowych, itd. Dane te mogą się bardzo różnić pod względem struktury, formatu i wielkości (mogą też nie posiadać żadnej struktury), co powinno być uwzględnione podczas tworzenia systemu big data.

- **Magazyn danych**: ta warstwa otrzymuje dane ze źródeł i jeśli to konieczne, przeprowadza konwersję do formatu, który może być zrozumiały przez narzędzia analityczne. Dane są następnie zapisywane do odpowiedniej bazy lub magazynu danych. Może to być relacyjna baza danych, choć często na tym etapie dane nie są uporządkowane w określony sposób lub nie posiadają konkretnej struktury, więc przechowuje się je w bazach NoSQL lub w rozproszonym magazynie plików, określanym często jako *data lake* (dosłownie: „jezioro danych”). Taki magazyn może pomieścić bardzo duże ilości plików w różnych formatach.

- **Warstwa analityczna**: przechowywane dane są przetwarzane w celu otrzymania z nich wartości biznesowej. Można tego dokonać za pomocą odpowiednich narzędzi do modelowania danych. Analiza może się odbywać automatycznie lub w postaci interakcyjnej eksploracji danych przez analityków i naukowców.

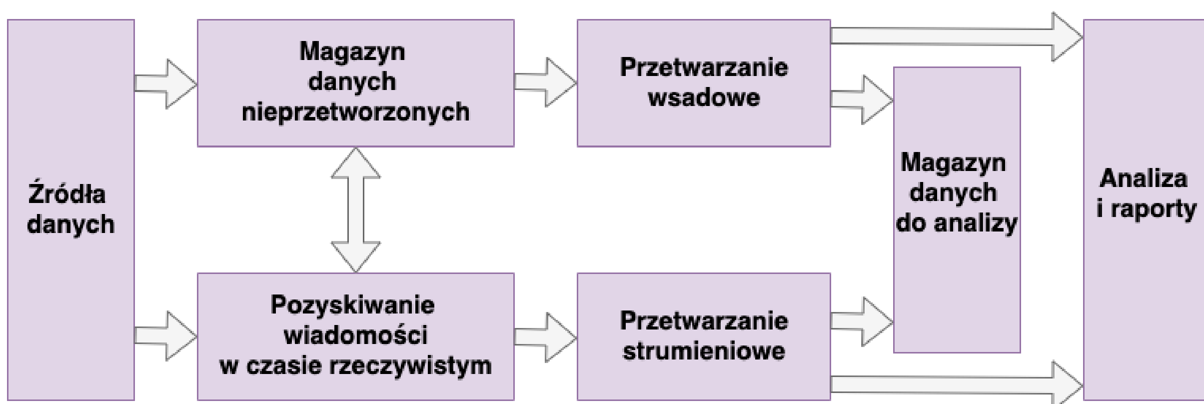
- **Warstwa konsumpcyjna**: czyli wyniki analizy przedstawione w odpowiedniej postaci wyjściowej – raportów, wykresów i wizualizacji, które są odpowiednie do przeglądania przez ludzi, ale też mogą służyć różnym aplikacjom.

Nieprzetworzone dane, które są przechowywane w bazach, mogą nie nadawać się od razu do analizy. Dlatego często architektura big data posiada również warstwę przetwarzania danych [6], w której dokonywane są odpowiednie transformacje, agregacje, konwersja i eliminacja niepotrzebnych elementów. W zależności od sposobu, w jaki dane są zbierane, wyróżnia się dwa rodzaje ich przetwarzania:

- **Przetwarzanie wsadowe**: polega na przetwarzaniu dużych bloków danych, które zostały zapisane do bazy w określonym przedziale czasu. Może to być tydzień, miesiąc, dzień – wszystko zależy od konkretnych potrzeb i oczekiwanych rezultatów. Dane, które są zbierane przez dłuższy czas mogą posiadać miliony rekordów i bardzo duże rozmiary. Narzędzia wykorzystywane do przetwarzania wsadowego muszą zatem być przystosowane do pracy z takimi plikami.

- **Przetwarzanie strumieniowe**: jest to przetwarzanie pojedynczych rekordów lub mikro partii danych w czasie rzeczywistym. Używa się go, gdy ważniejsze niż szczegółowa analiza jest otrzymywanie wyników możliwie najszybciej. W tym przypadku architektura musi również obejmować sposób na przechwytywanie i przechowywanie komunikatów w czasie rzeczywistym, a także jak najszybsze ich przetworzenie i analizę.

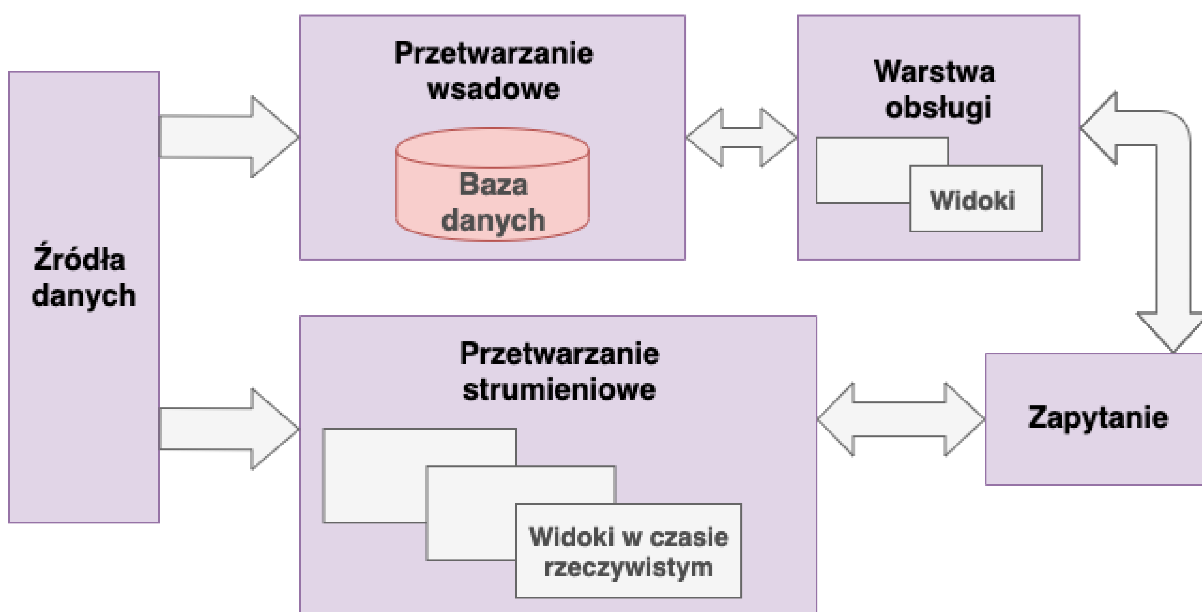
Schemat architektury zawierającej oba rodzaje przetwarzania zaprezentowany jest na rysunku 2.1. Widać też, że występuje tu podział warstwy magazynowej na dwa rodzaje: dane nieprzetworzone, napływające ze źródła (ang. *raw data*) oraz dane przetworzone (ang. *curated data*), z których korzystają narzędzia analityczne. Dane te mogą się znajdować w odrębnych bazach lub magazynach, a także w różnych kontenerach tego samego magazynu.



Rys. 2.1. Przykładowy schemat architektury big data. Źródło: opracowano na podstawie [6]

2.1.1. Architektura lambda

Potrzeba przetwarzania strumieniowego nie wyklucza chęci skorzystania z dokładniejszej i bogatszej analizy, którą oferuje przetwarzanie wsadowe. Decydując się na użycie obu sposobów przetwarzania, warto zastosować model architektoniczny, który zapewni jak najwięcej korzyści, skalowalność oraz ochronę przed błędami. Jednym z takich modeli jest architektura lambda [7], której schemat zaprezentowano na rysunku 2.2. Dane ze źródeł napływają zarówno do warstwy wsadowej, która je przetwarza partiami i tworzy widoki, jak i do warstwy szybkiej, w której najnowsze rekordy są od razu analizowane i przetwarzane na końcowe zapytania trafiające do użytkownika.



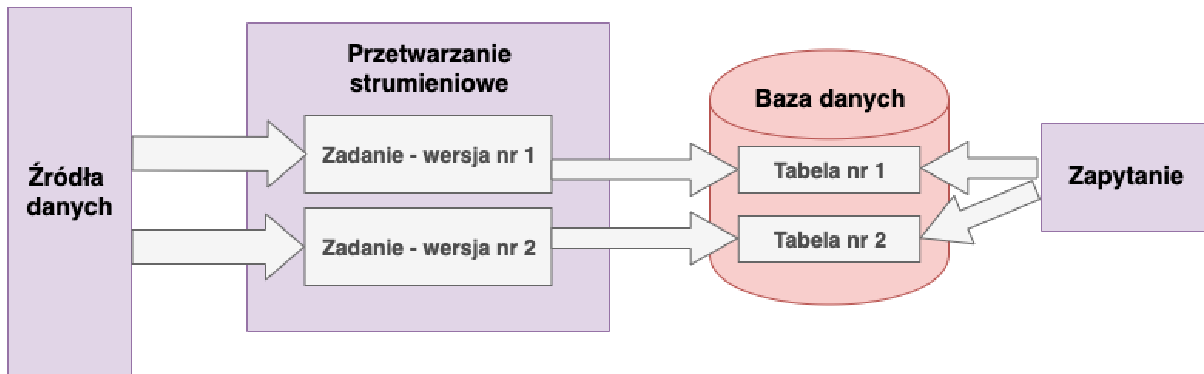
Rys. 2.2. Przykładowy schemat architektury big data. Źródło: opracowano na podstawie [7]

Zaletą takiej architektury jest posiadanie systemu analizującego dane niemal w czasie rzeczywistym, przy jednoczesnej możliwości wzbogacenia wyników takiej analizy o szczegółowe dane z określonego przedziału czasu. Wadą architektury lambda jest jej duża złożoność – każda część systemu musi posiadać odrębny kod i być dobrze zsynchronizowana z pozostałymi, by zapewnić spójność otrzymywanych wyników końcowych.

2.1.2. Architektura kappa

Problemy opisane powyżej nie występują w architekturze kappa [8]. Została ona stworzona jako uproszczona alternatywa dla architektury lambda. Oba modele mają bardzo podobne zastosowanie i cele – łączenie przetwarzania strumieniowego ze wsadowym. Różnicą jest to, że w architekturze kappa nie ma oddzielnego zestawu technologii dla części przetwarzania wsadowego, używa się tylko narzędzi do przetwarzania strumieniowego. Jak widać na rysunku 2.3,

dane ze źródeł przetwarzane są tylko w czasie rzeczywistym do postaci możliwej do analizy przez użytkownika, a następnie zapisywane do bazy. Przetworzone i zapisane dane mogą być w późniejszym czasie analizowane większymi partiami, by stworzyć dodatkowe, bardziej szczegółowe raporty.



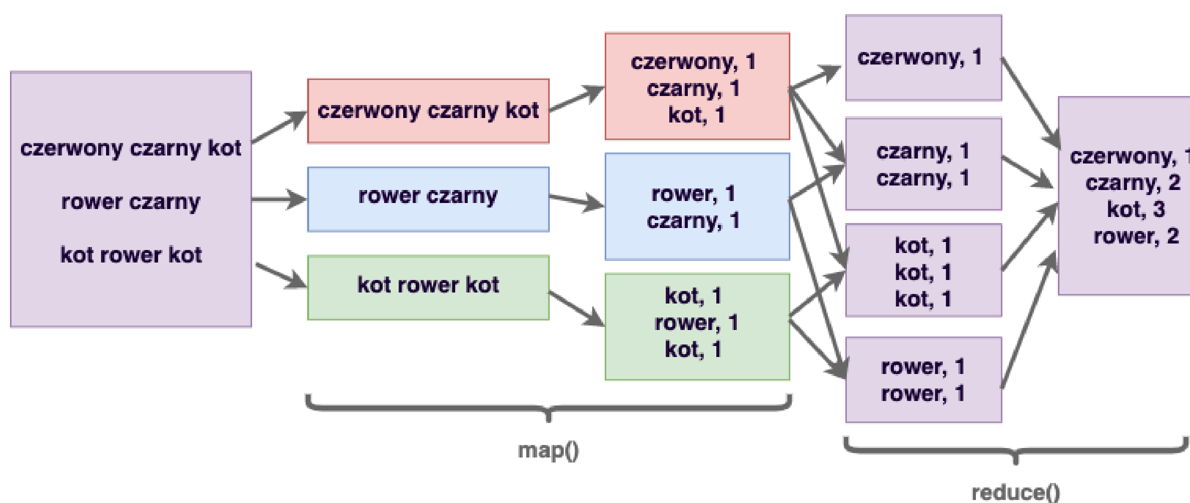
Rys. 2.3. Schemat architektury kappa. Źródło: opracowano na podstawie [8]

Architektura ta może nie sprawdzać się w przypadku, gdy mamy do czynienia z bardzo dużą ilością dużych danych. Narzędzia do przetwarzania wsadowego są zoptymalizowane do procesowania dużych ilości danych, więc są przypadki, gdy wykorzystanie ich jest konieczne do przeprowadzenia efektywnej i dokładnej analizy.

2.2 MapReduce

Wiele z obliczeń wykonywanych przez organizacje na etapie przetwarzania i analizy danych jest koncepcyjnie prostych, a ich największym problemem jest fakt, że dane wejściowe są bardzo duże. Aby móc otrzymać wyniki w rozsądnym czasie, konieczne jest rozdzielenie obliczeń między setkami lub tysiącami maszyn. Należy więc zastanowić się nad kwestią, jak najprościej zrównoleglić te obliczenia, rozdzielić dane i zachować przy tym niezawodność i obsługę błędów. Badacze z Google wzięli pod lupę to zagadnienie i zaprezentowali światu model programistyczny zwany MapReduce [9]. Ideą MapReduce jest podział problemu na zbiór zadań mapujących (ang. *map*), które na wyjściu dają parę klucz-wartość, oraz zadania redukującego (ang. *reduce*), które łączy wartości posiadające ten sam klucz.

Prześledźmy przykładowo problem zliczania wystąpień każdego słowa w dużym zbiorze dokumentów. Schemat działania MapReduce dla tego przypadku jest zaprezentowany na rysunku 2.4. Widzimy, że funkcja mapująca wczytuje tekst linijka po linijce i zwraca klucze – słowa oraz ich wartości, czyli liczbę wystąpień. Z racji tego, że na tym etapie nie wykonuje się żadnego sumowania, wartością słowa będzie zawsze 1. Następnie funkcja redukująca sumuje wystąpienia każdego słowa i zwraca ostatecznie listę wszystkich słów i ich wystąpień.



Rys. 2.4. Schemat działania MapReduce na przykładzie zliczania słów

MapReduce można też łatwo zastosować do takich problemów, jak na przykład: wyszukiwanie wyrażeń regularnych, częstotliwość odwiedzin danego adresu URL, rozproszone sortowanie czy uczenie maszynowe. Wiele z problemów klasyfikacyjnych można sprowadzić do zadania MapReduce. Przykładowo, w popularnym algorytmie k-średnich [10] każda iteracja to wczytywanie wejścia składającego się ze współrzędnych punktów reprezentujących dane i obliczenie dystansu od środków skupień. Każdy punkt zostaje przyporządkowany do środka skupień – klastra, który znajduje się najbliżej – otrzymujemy w efekcie parę klucz-wartość (klaster → współrzędne punktu). Następny etap to wyznaczenie średniej arytmetycznej współrzędnych wszystkich punktów należących do danego klastra. Współrzędne te stają się nowym środkiem skupień. Widzimy więc wyraźny podział na część mapującą, w której dochodzi do porządkowania i zwracania pośrednich par klucz-wartość oraz na część redukującą, w której wartości dla tego samego klucza są sumowane.

Zaletą MapReduce jest automatyczne dzielenie danych, zadań i obliczeń między maszynami. Poszczególne zadania mapujące i redukujące odbywają się równolegle, więc program jest skalowalny bez potrzeby dodawania dodatkowej implementacji.

2.2.1. Hadoop MapReduce

Po opublikowaniu paradygmatu MapReduce organizacja Apache Software Foundation, zajmująca się otwartym oprogramowaniem, stworzyła jego implementację, a także całe środowisko do rozproszonego przetwarzania i przechowywania plików, zwane Hadoop [11]. Kluczowymi składnikami środowiska są:

- **HDFS** (ang. *Hadoop Distributed File System*) [12]: jest to rozproszony system plików, bardzo odporny na błędy dzięki replikacji między węzłami, zapewniający wydajny dostęp do danych aplikacji, szczególnie w przypadku bardzo dużych zbiorów danych.

- **Hadoop YARN** [13]: środowisko uruchomieniowe pełniące rolę menadżera zasobów oraz monitorujące i planujące wykonywane zadania.
- **Hadoop MapReduce**: algorytm przetwarzający dane równolegle.

Główna zaleta środowiska Hadoop to możliwość przetwarzania dużych ilości danych naraz – z założenia jest więc dostosowane do przetwarzania wsadowego. Oparcie zasady działania na paradygmacie MapReduce sprawia jednak, że każdy przetwarzany problem musi się wpasować w ten model. Widać więc pewne ograniczenia tej technologii – rozważany problem może być trudny do sprowadzenia do zadań mapujących i redukujących lub wymagać wykonania serii takich operacji, jak na przykład wymieniony wcześniej algorytm k-średnich. Każda iteracja algorytmu jest oddzielnym zadaniem MapReduce, które musi być wykonywane sekwencyjnie, zatem czas wykonywania algorytmu jest wydłużony. Dodatkowo, przetwarzane dane są przechowywane bezpośrednio na dysku twardym, konieczne jest więc posiadanie szybkich i pojemnych dysków na maszynach.

2.3 RDD

Aby rozwiązać powyższe problemy, stworzono różne wyspecjalizowane frameworki, jak na przykład Pregel [14] – system do iteracyjnego przetwarzania grafów na dużą skalę, który trzyma pośrednie dane w pamięci operacyjnej, czy HaLoop [15] oferujący iteracyjny interfejs MapReduce. Wadą tych frameworków jest obsługa wyłącznie konkretnych wzorców. Potrzebna była abstrakcja, która mogłaby służyć do bardziej ogólnych przypadków, dlatego naukowcy z uniwersytetu w Berkeley stworzyli koncept „elastycznych rozproszonych zestawów danych" (ang. *resilient distributed datasets*), zwanych w skrócie RDD [16]. Są to niezmienne, odporne na błędy, działające równolegle struktury danych, które przechowują pośrednie wyniki w pamięci operacyjnej i pozwalają użytkownikowi na kontrolę nad podziałem danych.

Idea MapReduce jest zachowana – operacje mapowania i redukowania są częścią interfejsu RDD, jednak jest on rozszerzony również o wiele innych transformacji, jak na przykład filtrowanie, czy łączenie przetwarzanych równolegle struktur. Zamiast rzeczywistych danych, między poszczególnymi węzłami przekazywany jest zestaw operacji, jakie wykonywane są na strukturze RDD, co pozwala na efektywną obsługę błędów. Dzięki temu zgubione dane mogą być szybko odzyskane poprzez ponowne zastosowanie transformacji na tylko jednej partycji, bez konieczności przeprowadzania kosztownej replikacji całego zbioru danych.

2.3.1. Apache Spark

Do implementacji RDD stworzony został system Spark [17], który jest frameworkiem ogólnego przeznaczenia do analizy i przetwarzania dużych zbiorów danych. Umożliwia wydajne uruchamianie aplikacji zarówno dla danych przetwarzanych wsadowo, jak i strumieniowo. Według

twórców Spark wykonuje obliczenia 100 razy szybciej niż Hadoop (na przykładzie badania regresji logistycznej [17]). Zaletą jest również łatwość użycia – posiada bogaty interfejs, zintegrowany z różnymi językami, takimi jak Java, Scala, Python, R i SQL.

Aplikacje korzystające z systemu Spark mogą być uruchomione zarówno na natywnym klastrze, jak i na zewnętrznych klastrach, między innymi na środowisku YARN należącym do systemu Hadoop. Pozwala to na dużą swobodę w działaniu i konfiguracji, dzięki czemu łatwo zoptymalizować aplikację do potrzeb użytkownika.

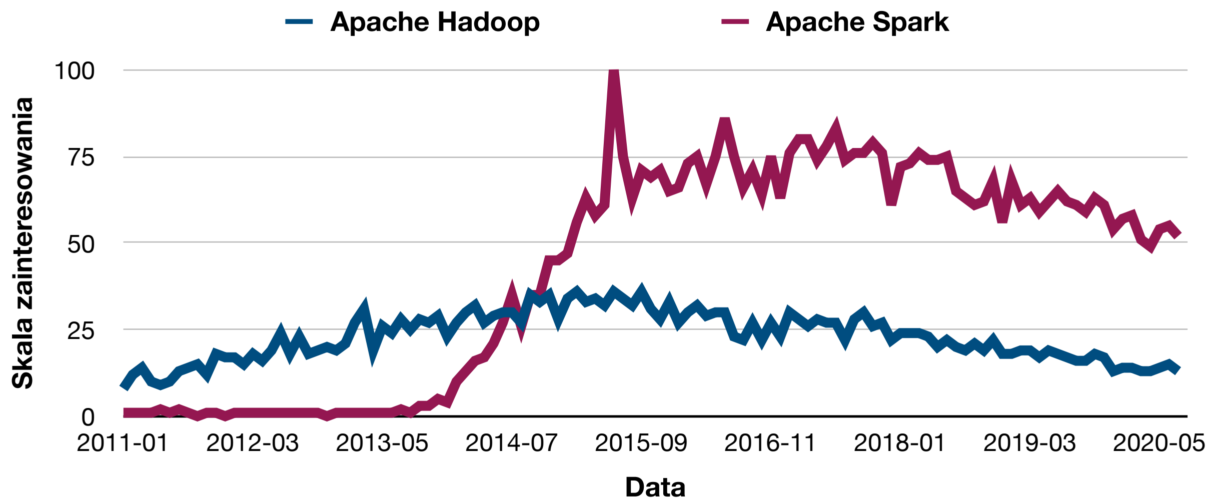
W wersji Apache Spark 2.2 i wyższych oprócz RDD znajdują się jeszcze dwie podobne struktury [18]: DataFrame i Dataset. DataFrame jest rozproszoną kolekcją danych zorganizowanych w nazwane kolumny – tak jak tabele w relacyjnej bazie danych. DataFrame posiada liczne operatory i agregacje oparte na wyrażeniach języka SQL, co ułatwia manipulację danymi. Struktura Dataset jest podobna do DataFrame i została wprowadzona, by zapewnić bezpieczeństwo typu (ang. *type safety*). Wszystkie błędy typów są znajdowane i zgłaszane na etapie kompilacji. Dzieje się to jednak kosztem wydajności – Dataset jest wolniejszy niż DataFrame.

Choć RDD jest najstarszą strukturą, nie oznacza to, że nie jest już wspierana, jednak jej wydajność i możliwości optymalizacyjne są dużo gorsze w przypadku przetwarzania danych o określonej strukturze.

2.4 Zmierzch technologii Hadoop

Technologia Hadoop została wprowadzona w 2006 roku i przez wiele lat było to wiodące rozwiązanie w dziedzinie big data. W ostatnich latach widać jednak, że rozwój architektury idzie wyraźnie w stronę przetwarzania danych w czasie rzeczywistym, dlatego wybierane są rozwiązania do tego dostosowane, czego przykładem jest Spark. Można więc stwierdzić, że technologia Hadoop zaczyna przeżywać swój schyłek. Jednym z argumentów za taką tezą jest analiza trendów w zainteresowaniu tematem „Hadoop” w wyszukiwarce Google (rysunek 2.5.). Widać, że wraz z wprowadzeniem technologii Spark, liczba wyszukiwań hasła „Apache Hadoop” spada i obecnie jest to jedynie 36% maksymalnego zainteresowania, które przypadło na rok 2015. Zainteresowanie technologią Spark niemal od początku było wyższe, niż to osiągnięte kiedykolwiek przez Hadoop, co może sugerować, że ta technologia ma więcej do zaoferowania, skoro cieszy się większym zainteresowaniem w wyszukiwarce.

Nasuwa się więc pytanie, czy w 2020 roku jest jeszcze powód do korzystania ze środowiska Hadoop, jak wypada ono w połączeniu z rozwiązaniami chmurowymi i czy artykuły głoszące jego kres [19] nie są czasem przedwczesne? Kwestia ta zostanie szerzej poruszona w dalszej części pracy.



Rys. 2.5. Popularność hasła w wyszukiwarce Google w ostatnich 9 latach.
Źródło: opracowano korzystając z [20]

3. Optymalizacja

Podczas przetwarzania big data można wyróżnić dwa rodzaje miar wydajności procesu [21]:

- **Miary efektywności**: czas przetwarzania; zużycie zasobów, czyli koszt procesu; pojemność – określająca maksymalną liczbę procesów, które mogą działać jednocześnie; czas oczekiwania od momentu przesłania polecenia wykonania zadania do momentu jego zakończenia; przepustowość, czyli ile zadań zakończyło się sukcesem w określonym przedziale czasu; opóźnienie, będące czasem, który z perspektywy użytkownika upływa do momentu aż otrzyma wyniki przetwarzanych danych; dokładność – wskaźnik błędów, które wystąpiły podczas przetwarzania.
- **Miary elastyczności**: czyli zdolność do reagowania na zmiany. Jest to: skalowalność, czyli możliwość przetwarzania danych, których rozmiar rośnie wraz z czasem; modyfikowalność – możliwość zmiany i dodawania nowych zadań lub narzędzi; możliwości konfiguracji, pozwalające na zmianę parametrów przetwarzania.

Widać więc, że chcąc mieć optymalny system big data, należy wziąć pod uwagę wiele czynników. Część z tych warunków może być spełniona już na etapie wyboru odpowiednich narzędzi do przetwarzania dużych ilości danych, jednak ważną kwestią jest również korzystanie z nich we właściwy sposób. Dołożenie kolejnego węzła do systemu czy zwiększenie pamięci podnosi ogólny koszt operacji, a niekoniecznie gwarantuje lepszą wydajność przetwarzania. Dlatego przedmiotem tej pracy jest analiza optymalizacji na poziomie aplikacji, podziału i przechowywania danych oraz wykorzystania istniejących zasobów.

3.1 Metodologia pomiarów

Poszczególne rozwiązania opisywane w niniejszej pracy badane są pod kątem czasu, zużycia pamięci, skalowalności oraz łatwości użycia i konfiguracji. Zarówno Spark, jak i Hadoop posiadają wbudowany interaktywny monitoring zadań, w którym również zawarte są szczegółowe metryki na temat poszczególnych uruchomień. Kwestia odporności na błędy nie jest rozpatrywana, gdyż analizowane technologie w założeniu mają zaimplementowane mechanizmy replikacji lub przekazywania informacji między węzłami.

Pomiar czasu jest główną braną pod uwagę metryką wydajności. Z założenia zadanie, które wykona się szybciej, zużyje też sumarycznie mniej zasobów i będzie mniej kosztowne. Czas w analizowanych przypadkach rozpatrywany jest ogólnie, jako całkowity czas wykonania zadania, jak i szczegółowo, jako czas wykonywania konkretnych fragmentów lub funkcji programu. Wiele niezależnych czynników może wpłynąć na czas działania poszczególnego programu, dlatego porównano wiele uruchomień tej samej aplikacji. Dołożono wszelkich starań, by każde uruchomienie tego samego programu odbywało się w identycznych warunkach, dzięki czemu otrzymywane wartości były zbliżone do siebie, jednak mimo to występowały pewne elementy odstające. Odrzucone zostały czasy większe niż 50% typowej otrzymywanej wartości. Jednak nadal pojawiały się wartości, które trudno było jasno zakwalifikować jako elementy nietypowe, ale wpływały na wyznaczoną średnią arytmetyczną. Dlatego aby otrzymać typowy, odporny na elementy odstające czas wykonania zadania, użyta została mediana.

Hadoop został napisany w języku Java, a Spark w języku Scala, zatem oba frameworki uruchamiane są na wirtualnej maszynie Javy. Duży wpływ na wydajność aplikacji ma więc czas działania *garbage collectora* [22]. Metryka ta często pozwala na wykrycie problemów związanych z programem – złe zarządzanie pamięcią jest szczególnie uciążliwe w przypadku dużych ilości danych i może nawet prowadzić do tego, że dojdzie do przepełnienia stosu i zawieszenia działania uruchomionej aplikacji.

Zużycie pamięci mierzone jest w $\text{MB} \cdot \text{s}$ i opisuje całkowitą zaalokowaną przez aplikację pamięć w megabajtach, pomnożoną przez czas działania aplikacji w sekundach. Dodatkowo mierzona jest liczba tzw. wirtualnych rdzeni użytych do wykonania zadania. Pokazuje ona efektywność alokacji pamięci – mniej wirtualnych rdzeni oznacza lepsze wykorzystanie dostępnych zasobów.

Wymieniona wcześniej miara łatwości użycia i konfiguracji może być trudna do opisania w matematycznym ujęciu. Liczba linii kodu poszczególnych programów, dających ten sam efekt wyjściowy, umożliwia w pewnym stopniu porównanie ich skomplikowania. Nie można oczywiście zakładać, że mniej linii kodu oznacza, że program jest automatycznie łatwiejszy w implementacji, ale w omawianych w tej pracy technologiach ta zależność często jest widoczna. Przykładowo, w Hadoop MapReduce, aby zsumować wartości dla danego klucza, potrzebna jest implementacja klasy rozszerzającej klasę `Reducer` oraz implementacja metody `reduce()` i zawarcie w niej pętli sumującej wszystkie wartości dla danego klucza. Korzystając z RDD w bibliotece Spark, można osiągnąć to samo przy pomocy gotowej już funkcji `reduceByKey()`. Dodatkowo mniejsza złożoność programu powoduje, że jest on łatwiejszy do utrzymania i wprowadzania zmian.

4. Wykorzystane technologie

4.1 Ekosystem Hadoop

Środowisko Hadoop i jego podstawowe moduły zostały przybliżone w rozdziale 2.2.1. Termin Hadoop jest jednak bardzo często używany w kontekście tzw. ekosystemu, składającego się z licznych modułów i powiązanych projektów. Wiele z nich powstało jako część projektu Apache Hadoop, a potem uniezależniło się i rozwinęło w nowym kierunku. Dlatego trudno jest określić dokładną liczbę modułów w tym ekosystemie.

W niniejszej pracy skorzystano z technologii Hadoop, a także z dwóch powiązanych z nią projektów rozszerzających możliwości bazodanowe: HBase oraz Hive.

4.1.1. HBase

Apache HBase [23] to nierelacyjna, rozproszona baza danych przeznaczona dla big data. Została stworzona na potrzeby przechowywania bardzo dużych tabel o miliardach wierszy i milionach kolumn. Baza HBase jest uruchamiana na rozproszonym systemie plików HDFS i rozszerza jego funkcjonalność o indeksowanie danych i możliwość bezpośredniego dostępu do nich poprzez zapytania. Jej podstawowymi cechami są: duża liniowa skalowalność, brak języka zapytań, przechowywanie danych w postaci klucz-wartość oraz małe opóźnienie w dostępie do pojedynczych rekordów, nawet w przypadku miliardów wierszy – dobrze działa więc dla zapytań w czasie rzeczywistym.

HBase to baza zorientowana kolumnowo – oznacza to, że dane przechowywane są jako sekcje kolumn, co dobrze sprawdza się w przypadku bardzo dużych tabel, wykorzystywanych w celach analitycznych. Chcąc przykładowo wyznaczyć liczbę sprzedanych produktów we wszystkich sklepach, baza kolumnowa odczytuje tylko kolumnę związaną z kosztem i statusem produktu, podczas gdy tradycyjna relacyjna baza przegląda wszystkie dane dla każdego wiersza. W HBase schemat tabeli definiowany jest wyłącznie poprzez rodziny kolumn, w których może się znajdować wiele kolumn będących zbiorem par klucz-wartość.

4.1.2. Hive

Kolejną technologią bazodanową stworzoną, by rozszerzyć możliwości systemu Hadoop, jest Apache Hive [24]. Jest to hurtownia danych, umożliwiająca ich analizę i podsumowanie, jak również przetwarzanie zapytań dla dużych zbiorów niezorganizowanych danych. Hive nie jest bazą danych, ale abstrakcją, która zapewnia interfejs do wykonywania zapytań w języku SQL. Może być więc używany w połączeniu z plikami przechowywanymi w systemie HDFS, a także jako warstwa tabeli i języka zapytań w HBase. Posiada również dobrą integrację z Apache Spark oraz wieloma narzędziami do analityki biznesowej.

4.2 Spark

Platforma Apache Spark została opisana w rozdziale 2.3.1. Wiele źródeł wymienia ją jako część ekosystemu Hadoop [25], lecz stwierdzenie to może nasuwać mylne wrażenie, że jest to moduł rozszerzający, a nie samodzielny projekt. Spark posiada własny system zarządzania klastrami, ale może być równie dobrze uruchomiony na klastrze Hadoop YARN, Mesos czy Kubernetes [26].

Spark powstał jako odpowiedź na ograniczenia MapReduce i używa się go w systemach korzystających zarówno z przetwarzania wsadowego, jak i strumieniowego. Posiada wiele bibliotek, takich jak między innymi: Spark SQL do przetwarzania danych uporządkowanych, Spark Streaming do danych napływających w czasie rzeczywistym, MLlib do uczenia maszynowego czy GraphX będący narzędziem do obliczania grafów. W tej pracy wykorzystane zostaną biblioteki Spark SQL i Spark Streaming.

4.3 Cassandra

Apache Cassandra [27] jest, tak samo jak HBase, bazą NoSQL zorientowaną kolumnowo i nastawioną na zarządzanie bardzo dużymi ilościami danych. Główną różnicą jest możliwość wykonywania zapytań w języku CQL będącym pochodną SQL. Cassandra jest samowystarczalną technologią do zarządzania i przechowywania danych i nie jest zależna od żadnych innych modułów. Jej cechy to odporność na błędy, wysoka wydajność, skalowalność, trwałość i elastyczność. Dodatkowo, Cassandrę łatwo połączyć z frameworkiem Spark i takie też jest jej zastosowanie w niniejszej pracy.

4.4 Microsoft Azure

Choć big data może istnieć samodzielnie, na lokalnej infrastrukturze, to jednak nie powinno się ignorować licznych korzyści, jakie przynosi przeniesienie tych technologii do chmury obliczeniowej. Chcąc przetwarzać dane na ogromną skalę oraz mieć system, który łatwo i szybko można

dostosować do rozwijających się potrzeb organizacji, najlepiej zainwestować w subskrypcję u jednego z dostawców. Obecnie czołowymi serwisami [28] są: Microsoft Azure, Amazon Web Services, Salesforce i Google Cloud.

W tej pracy skorzystano z platformy Microsoft Azure [29], gdyż posiada bardzo rozbudowany zestaw usług, dobrą obsługę narzędzi do big data, w tym Hadoop i Spark, bogatą i łatwą w użyciu dokumentację oraz całkowicie darmową miesięczną wersję próbną, co jest dużym atutem w przypadku projektu studenckiego. W ramach chmury Azure skorzystano z usługi HDInsight [30] oraz Databricks [31]. Jako magazyn danych wykorzystana została usługa Azure Blob Storage [32], służąca do przechowywania nieustrukturyzowanych danych na dużą skalę.

W usłudze HDInsight można uruchomić popularne narzędzia otwartego oprogramowania, takie jak Hadoop, Spark, HBase, Hive i Kafka. Umożliwia ona szybkie i proste wdrożenie oraz skalowanie klastrów danych big data. Udostępnia również liczne narzędzia do monitorowania i analizy danych.

4.4.1. Databricks

Chmura Azure posiada obsługę Databricks – wielozadaniowej platformy do przetwarzania i obsługi danych, uczenia maszynowego i analizy statystycznej. Serwis ten został założony przez twórców Apache Spark i jest w całości oparty na tej technologii. Zaletą Databricks jest łatwość użycia – można tworzyć i uruchamiać programy w interaktywnym obszarze roboczym lub bezpośrednio na klastrze Apache Spark. Platforma jest zoptymalizowana, aby zapewnić maksymalną wydajność – umożliwia przetworzenie i analizę terabajta danych w kilka minut [33]. Integracja Databricks z narzędziami Azure umożliwia stworzenie pełnego systemu przetwarzania big data.

5. Przykład obliczeniowy

Aby mieć wstępnie ogólny obraz działania analizowanych technologii, zbadane zostały różnice w ich wydajności obliczeniowej. Do testów użyty został prosty algorytm wyznaczania wartości liczby π metodą Monte Carlo.

Obliczenia lokalne wykonane zostały na maszynie posiadającej 4 GB pamięci RAM, dysk SSD 256 GB i dwurdzeniowy procesor Intel Core i5 1,4 GHz. Do obliczeń w chmurze Azure wykorzystano klastry HDInsight i Databricks z trzema 8-rdzeniowymi węzłami z 28 GB pamięci RAM każdy, procesorem Intel Xeon 8171M 2.1 GHz oraz dyskami SSD 400 GB.

5.1 Scala a Java

Framework Hadoop MapReduce został zaimplementowany w Javie, natomiast Spark w Scali, dlatego te właśnie języki zostały użyte przy pisaniu testowych aplikacji w niniejszej pracy. Na początek warto zatem przeanalizować różnice w ich wydajności.

Zarówno Java, jak i Scala działają na wirtualnej maszynie Javy (JVM). Kod źródłowy jest kompilowany do kodu bajtowego, więc z perspektywy JVM nie ma żadnej różnicy między tymi językami. Jeśli więc uruchomimy w Scali i Javie kod o niemal identycznej składni, czas działania programu powinien być bardzo podobny.

Listingi 1. i 2. przedstawiają implementację algorytmu wyznaczania liczby π kolejno w Javie i Scali. Widzimy, że oba programy nie różnią się zbytnio pod względem składni – można się więc spodziewać podobnych wyników czasowych. Dla każdego z nich wykonano 10 prób, dla $n = 10000$.

```java
double x, y;
int count = 0;
for (int i = 1; i <= n; i++) {
    x = Math.random() * 2 - 1;
    y = Math.random() * 2 - 1;
    if (x * x + y * y <= 1) count++; }
double pi = 4. * count / (n - 1);
```

Listing 1. Wyznaczanie liczby π – implementacja w języku Java

```
var count: Int = 0
var i: Int = 1
while (i <= n) {
  val x = random * 2 - 1
  val y = random * 2 - 1
  if (x * x + y * y <= 1) count += 1
  i += 1 }
val pi: Double = 4.0 * count / (n - 1)
```

Listing 2. Wyznaczanie liczby π – implementacja w języku Scala

Mediana czasu obliczeń dla Javy wyniosła 0,87 s, natomiast dla Scali – 0,85 s. Wyniki te są na tyle zbliżone do siebie, że można założyć, że przy większej liczbie prób ta różnica czasowa stałaby się pomijalna. Po co więc istnieje Scala skoro jej kod jest tak zbliżony do kodu napisanego w Javie? Otóż większość programistów Scali uznałaby listing 2. za „brzydki" kod i zapisałaby ten algorytm w formie bardziej idiomatycznej. Scala pozwala na używanie programowania funkcyjnego [34], dzięki czemu czemu kod można zapisać w bardziej zwięzłej formie. Taki zapis przedstawia listing 3. Widzimy, że linijek jest mniej, nie ma typowej pętli oraz zostały użyte funkcje `map()` i `reduce()`. Nasuwa się tu skojarzenie z MapReduce i choć Scala nie ma związku z tym paradygmatem, to jest to kolejny dowód, że mapowanie i redukowanie może zastąpić zwykłą pętlę iteracyjną.

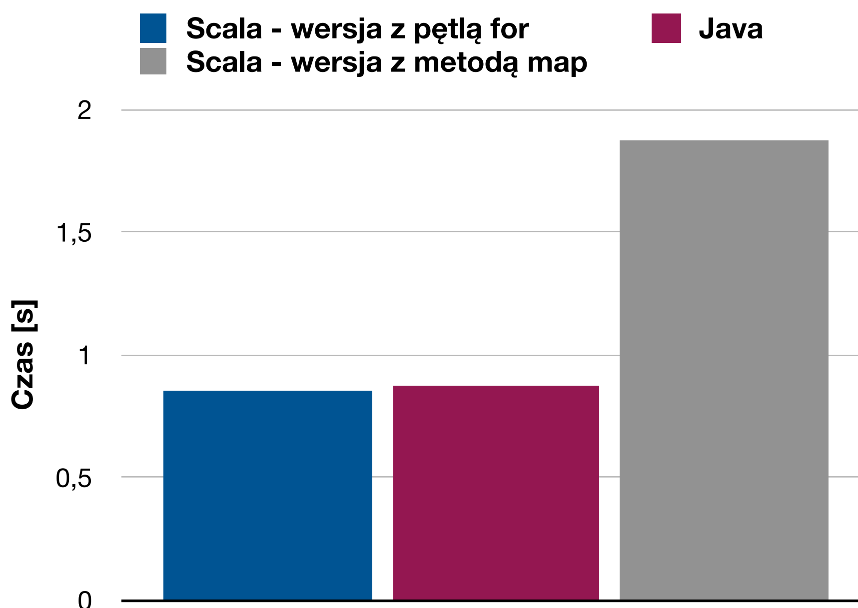
```
val pi = (1 until n).map { _ =>
  val x = random * 2 - 1
  val y = random * 2 - 1
  if (x * x + y * y <= 1) 1 else 0
}.reduce(_ + _) * 4.0 / (n - 1)
```

Listing 3. Wyznaczanie liczby π – implementacja w języku Scala, styl idiomatyczny

Mediana dla 10 uruchomień powyższego kodu wyniosła 1,88 s. Wykres 5.1. przedstawia porównanie wyników – widać, że zapisanie kodu w idiomatycznym stylu Scali wydłuża czas około dwukrotnie.

Ponownie można sobie więc zadać pytanie – po co używać Scali, skoro kosztem „ładniejszego" kodu mamy gorszą wydajność programu? Trzeba pamiętać, że narzędzia dostosowuje się do potrzeb, a nie odwrotnie. I faktycznie, wyznaczanie liczby π czy inne problemy obliczeniowe nie są najlepszym zastosowaniem języka Scala. Żeby zobaczyć korzyści płynące ze Scali, trzeba by sprawdzić jej działanie w bardziej praktycznym zastosowaniu. Scala, jak już sama jej nazwa mówi,

zostało stworzona jako język skalowalny. Zapewnia łatwy i bezpieczny paralelizm, a dodatkowo zwięzły kod jest lepszy do utrzymania i rozszerzania.



Rys. 5.1. Porównanie mediany wyników

5.2 Hadoop i Spark lokalnie

Sama analiza języków programowania nie mówi wiele o wydajności technologii Hadoop i Spark, ale otrzymane wyniki będą służyły jako punkt odniesienia. Można się spodziewać, że użycie do problemu obliczeniowego frameworku do przetwarzania danych da gorszy wynik niż program napisany po prostu w języku Java lub Scala. Sprawdźmy więc, czy tak jest w rzeczywistości.

Algorytm wyznaczania liczby π zaimplementowano w języku Java przy użyciu Hadoop MapReduce i uruchomiono 10 razy dla 10 000 000 iteracji na maszynie lokalnej. Kod programu różni się znacznie od tego analizowanego w poprzednim podrozdziale, gdyż oprócz pętli wyliczającej liczbę π, konieczne było również skonfigurowanie zadania dla klastra Hadoopa oraz dołożenie zapisu pośrednich wyników do pliku – Hadoop nie wykonuje żadnych obliczeń w pamięci operacyjnej i korzysta jedynie z danych zapisywanych na dysku twardym.

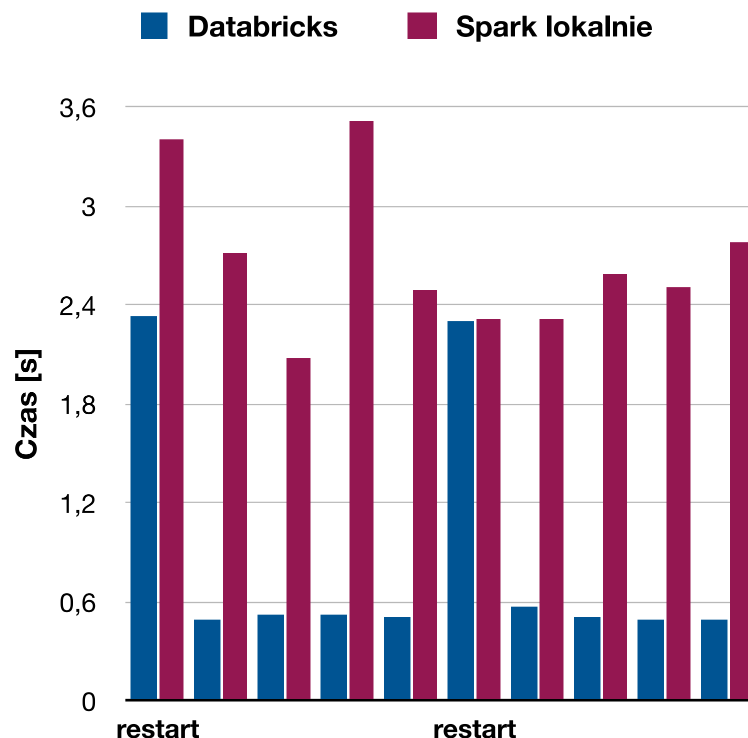
Implementację z wykorzystaniem Sparka i języka Scala uruchomiono 10 razy po $5 \cdot 2\,000\,000$ iteracji, gdzie 5 jest liczbą równolegle działających funkcji mapujących. Hadoop też może równolegle wykonywać zadania mapujące, ale nie jest to możliwe do osiągnięcia lokalnie na środowisku pseudorozproszonym, bo obliczenia pośrednie są wykonywane i zapisywane do pliku na dysku – dlatego mając jeden dysk, 5 zadań mapujących wykonałoby się po kolei. Kod napisany w Sparku korzysta w niemal niezmienionej formie z listingu 3., dołożony jest jedynie obiekt `SparkContext`, który zrównolegla obliczenia w klastrze.

Mediana czasu wykonania algorytmu dla Sparka wyniosła 2,55 s, natomiast dla Hadoopa aż 33 s, czyli niemal 13 razy wolniej. Różnica jest ogromna, ale spodziewana – w rozdziale 2.3.1. szybkość obliczeniowa została wymieniona jako jedna z przewag Sparka nad Hadoopem. Przykład napisany w Sparku uruchomiono również bez zrównoleglania obliczeń i w tym przypadku mediana czasu działania wyniosła 3,43 s.

5.3 Hadoop i Spark w chmurze

Kolejnym etapem było uruchomienie na chmurze Azure tego samego programu, co lokalnie. Dzięki temu zobaczymy, czy obliczenia wykonane na kilku lepszych maszynach dadzą inny wynik.

Pierwsze uruchomienie programu na klastrze Databricks dało wynik 2,33 s, czyli porównywalny do tego otrzymanego lokalnie. Widać więc, że więcej pamięci RAM i węzłów nie ma większego znaczenia w przypadku mało wymagającego problemu. Ciekawsze było drugie uruchomienie algorytmu na klastrze – czas obliczeń wyniósł tylko 0,5 s. Na wykresie 5.2. przedstawione są wyniki z poszczególnych prób dla klastra Databricks i lokalnego klastra Spark. Widzimy, że po restarcie klastra w Databricks, czas wykonania obliczeń jest znacznie dłuższy niż czas w każdej kolejnej próbie.

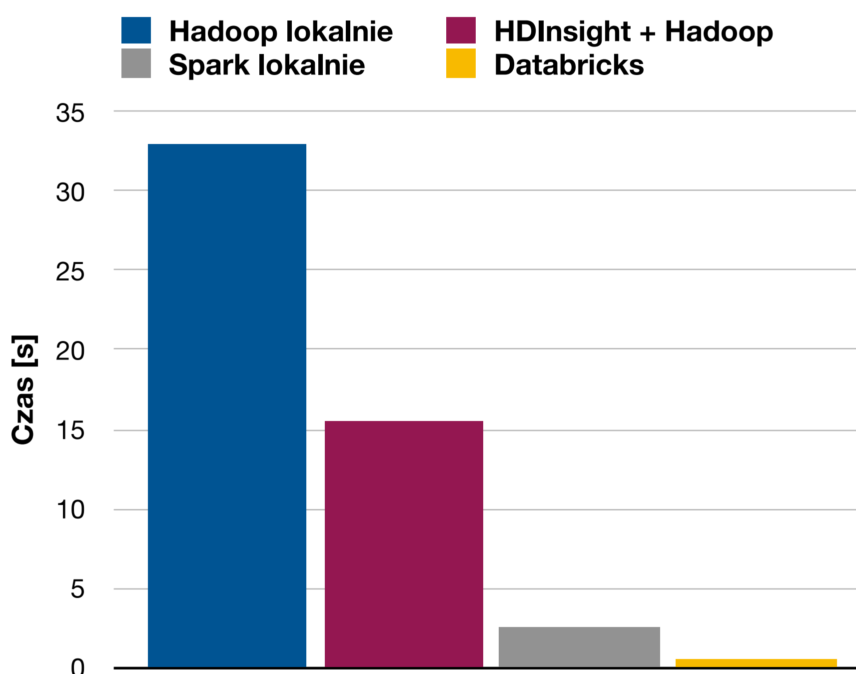


Rys. 5.2. Porównanie czasu obliczeń dla kolejnych uruchomień, z restartem klastra Databricks po 5. uruchomieniu

Ostatecznie więc mediana z 10 prób wyniosła 0,51 s, czyli mniej, niż kod napisany w zwykłej pętli w Javie. Dzieje się tak dlatego, że Databricks automatycznie zapisuje do pamięci podręcznej

dane ze struktury RDD, co pozwala na znaczne skrócenie czasu działania przy drugim i kolejnym uruchomieniu programu na klastrze. Mechanizm pamięci podręcznej [35] można także włączyć na klastrze Sparka za pomocą dodania funkcji `cache()` do programu. Jednak w testowanym przykładzie nie przyniosło to korzyści – czas obliczeń w kolejnych uruchomieniach na lokalnym klastrze miał zbliżoną wartość do czasu obliczeń bez zapisu do pamięci podręcznej. Klaster Databricks przechowuje pamięć podręczną na dysku, natomiast zwykły klaster Sparka – w pamięci RAM, więc zapisane dane zajmują miejsce, które mogłoby być wykorzystane na obliczenia. Dlatego nie ma poprawy po dodaniu tej opcji.

Ostatnim analizowanym w tym przykładzie środowiskiem jest klaster Apache Hadoop na platformie Azure HDInsight. Mediana czasu obliczeń dla 10 uruchomień wyniosła 15,5 s. Widać więc znaczną poprawę w stosunku do lokalnego klastra, ale nadal jest to bardzo zły w wynik w porównaniu z czasem osiąganym przez program napisany przy użyciu frameworku Spark. Na wykresie 5.3. przedstawiono wyniki dla technologii analizowanych w tym rozdziale. Widzimy, że dla problemów obliczeniowych przewaga Databricks jest ogromna, a czas obliczeń ponad 30-krotnie mniejszy niż dla programu korzystającego z Hadoop MapReduce, uruchomionego na klastrze o identycznych parametrach.



Rys. 5.3. Porównanie mediany czasu obliczeń dla poszczególnych technologii

6. Architektura lokalna

W tym rozdziale prezentowane będzie działanie programów uruchomionych lokalnie na komputerze (parametry wymienione zostały na początku rozdziału 5.) w środowisku pseudorozproszonym, czyli z tylko jednym węzłem. Konfigurowanie poszczególnych klastrów na lokalnej maszynie pozwala mieć lepszy wgląd w ich działanie, dodatkowo daje swobodę w łączeniu różnych technologii i baz danych, co ułatwia przeprowadzenie bardziej szczegółowego porównania.

Wykorzystane zostały następujące technologie i ich wersje:

- **Apache Hadoop**: 3.2.1.
- **Apache Spark**: 2.3.2.
- **Apache Cassandra**: 3.11.6.
- **Apache HBase**: 1.3.5.

Analizowane technologie zostały stworzone do działania w architekturze rozproszonej. Skalowalność oraz możliwość łatwego zrównoleglenia obliczeń i operacji to ich podstawowe cechy, dlatego te aspekty będą omawiane w następnym rozdziale, dotyczącym architektury w chmurze.

6.1 Przetwarzanie danych nieustrukturyzowanych

Dane nieustrukturyzowane to takie, które nie posiadają żadnego określonego schematu – mają często formę plików tekstowych, prezentacji, stron internetowych czy e-maili. Przetwarzanie takich danych często polega na wyszukaniu w nich określonego wzorca i grupowaniu w kategorie.

W tej części analizowany będzie przykład zliczania słów, który został opisany w rozdziale 2.2. Zbiór danych to 418 książek różnej długości, pobranych z serwisu Wolne Lektury [36].

6.1.1. Klaster Apache Hadoop

Pierwsza implementacja przykładu wykorzystuje framework Hadoop MapReduce, natomiast dane przechowywane są w rozproszonym systemie plików HDFS, z którego korzysta klaster.

Hadoop przechowuje dane w tzw. blokach [37] o domyślnym rozmiarze 128 MB. Odczyt i zapis w ramach tego samego bloku wymaga wysłania jednego wstępnego żądania do węzła głównego,

co przyspiesza przetwarzanie danych. Nie jest więc przystosowany do przetwarzania dużej liczby małych plików, gdyż każdy plik przypisywany jest do oddzielnego bloku danych. Każdy blok alokuje 150 bajtów na stosie pamięci węzła głównego [38], więc mając bardzo dużo małych plików, łatwo doprowadzić do przepełnienia stosu. Do tego każdy blok danych jest obsługiwany przez oddzielne zadanie mapujące, co znacznie wydłuża czas działania programu. Mając więc 418 książek zapisanych w plikach tekstowych, będziemy mieć 418 zadań mapujących. Uruchomienie programu potwierdza te przypuszczenia – całkowity czas wykonania zadania to 37 min 3 s (ze względu na skalę tego wyniku, przeprowadzona została tylko jedna próba). Jest to wynik bardzo zły, szczególnie że całkowity rozmiar przetwarzanych danych to tylko 40,2 MB.

W tym przypadku najszybszym rozwiązaniem będzie połączenie wszystkich plików w jeden. HDFS posiada komendę `-getmerge`, za pomocą której można to osiągnąć w prosty sposób. Cały połączony plik ma rozmiar 39,3 MB, więc mieści się w jednym bloku danych i w efekcie wykona się tylko jedno zadanie mapujące. Tabela 6.1. przedstawia medianę wyników z 10 uruchomień. Całkowity czas działania aplikacji to tzw. czas oczekiwania, liczony od momentu przesłania polecenia wykonania aplikacji do jej zakończenia – zawiera więc również czas poświęcony na przyjęcie programu przez klaster, alokację bloków pamięci oraz dodanie do kolejki i przyjęcie przez menadżera zasobów YARN. Z tego powodu zmierzony został również czas samego wykonania programu, czyli od momentu startu pierwszego zadania mapującego, do momentu zapisania ostatecznego wyniku.

Całkowity czas	44,5 s
Czas wykonania programu	34,5 s
Czas mapowania	21 s
Czas redukowania	2 s
Czas działania *garbage collectora*	0,19 s
Zużycie pamięci	233 216,5 MB · s
Liczba wirtualnych węzłów · całkowity czas	80 · 1 s

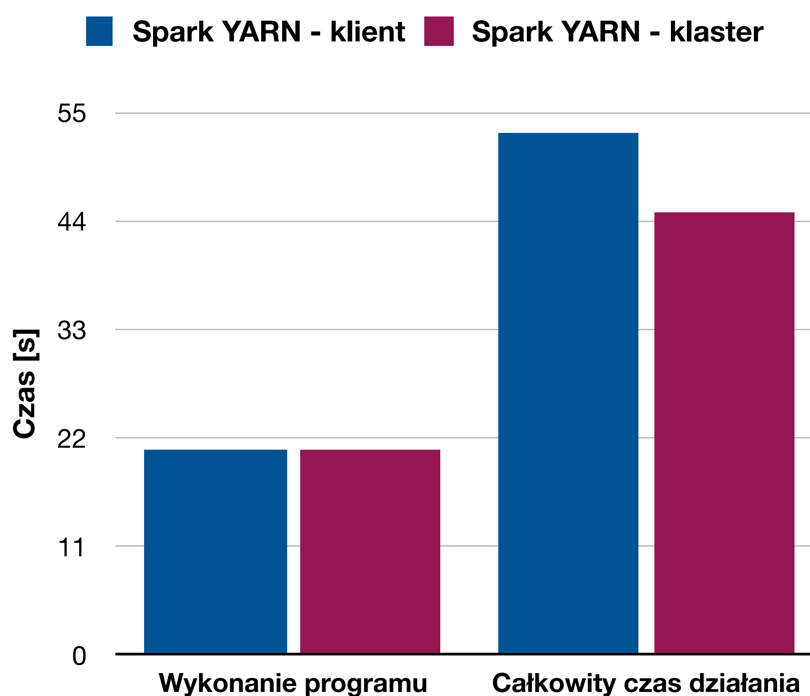
Tabela 6.1. Mediana dla 10 uruchomień programu wykorzystującego Hadoop MapReduce, z danymi połączonymi w jeden plik

Widzimy, że połączenie plików znacznie poprawiło czas działania. W kwestii optymalizacji tego programu już niewiele można osiągnąć, czas mapowania mógłby się poprawić wyłącznie mając lepszy sprzęt. Wynik zliczania słów, w postaci klucz-wartość, został zapisany do pliku tekstowego. Hadoop MapReduce automatycznie sortuje dane alfabetycznie po kluczu, co ułatwia przegląd pliku wynikowego.

Na tym samym klastrze można uruchomić program napisany przy użyciu frameworku Spark. Istnieją dwa sposoby [39] wdrożenia zewnętrznej aplikacji do menadżera zasobów Hadoop YARN.

Pierwszy z nich to tryb klastra, w którym sterownik Sparka uruchamiany jest w głównym procesie aplikacji i jest odpowiedzialny za wysyłanie żądań o zasoby do menadżera YARN. Drugi sposób uruchomienia to tryb klienta, w którymi sterownik działa w oddzielnym procesie. Nadaje się on dobrze do interaktywnego korzystania z programu, gdy potrzebna jest komunikacja z użytkownikiem. Podstawowe różnice pomiędzy tymi trybami nie pozwalają na określenie, który z nich będzie optymalny pod względem wydajności, dlatego w celu porównania program został uruchomiony w obu trybach.

Na wykresie 6.1. zaprezentowane jest porównanie uruchomień tej samej aplikacji w trybie klienta i klastra. Sam czas wykonania programu uzyskano niemal identyczny, co potwierdza, że wybór sposobu uruchomienia nie wpływa w żaden sposób na wewnętrzny sposób wykonywania aplikacji. Różnice występują dopiero w momencie, gdy przeanalizujemy całkowity czas trwania zadania. W trybie klastra program zakończył się średnio o 8 s wcześniej niż w trybie klienta. Analiza metryk poszczególnych uruchomień pokazała, że w trybie klienta sterownik Sparka działa bezczynnie przez kilka sekund, zanim zostanie zarejestrowany na klastrze Hadoopa.

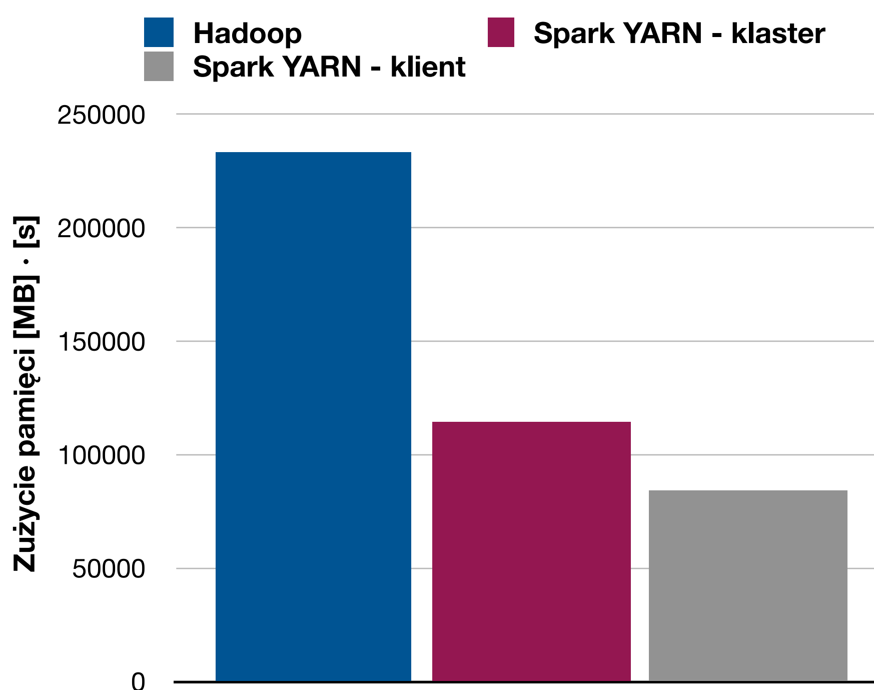


Rys. 6.1. Mediana dla 5 uruchomień programu, z danymi połączonymi w jeden plik

Powyższe wyniki przedstawiają uruchomienie aplikacji z domyślną konfiguracją. Sterownik Sparka ma dużo parametrów i opcji, które można samodzielnie dostosować, więc warto się zastanowić, czy możliwe jest wprowadzenie pewnych ulepszeń. Analizując logi aplikacji można zauważyć, że wykonanie zadania zostało rozdzielone pomiędzy dwa egzekutory. Liczba egzekutorów domyślnie odpowiada sumarycznej liczbie rdzeni na wszystkich węzłach, więc mając tylko jedną maszynę z dwurdzeniowym procesorem, zadanie zostanie podzielone na dwie równolegle

wykonujące się części. Jednak dla małego rozmiaru danych, wprowadzenie takiej równoległości może być zbędne i jest też niepotrzebnym marnowaniem zasobów. I rzeczywiście, po uruchomieniu aplikacji z jednym tylko egzekutorem, całkowity czas działania w trybie klastra skrócił się do 33 s, natomiast w trybie klienta do 45 s. Zużycie pamięci w trybie klastra z 212 462 MB · s zmniejszyło się do 114 623 MB · s, natomiast w trybie klienta z 161 487 MB · s zmniejszyło się do 83 588 MB · s, więc w obu przypadkach spadło niemal o połowę. Natomiast czas wykonywania programu do zliczania słów wyniósł 17 s w obu przypadkach.

Porównując powyższe wyniki z czasem działania aplikacji korzystającej z Hadoop MapReduce, widać przewagę kodu napisanego przy użyciu frameworku Spark w trybie jednego egzekutora. Wykres 6.2. pokazuje różnice w wykorzystaniu pamięci w poszczególnych wariantach. Aplikacja Spark działająca w trybie klienta, choć miała najdłuższy czas trwania, okazała się też bardziej oszczędna w kwestii alokacji pamięci. Najgorzej w tym zestawieniu wypada aplikacja Hadoop, która alokuje znacznie więcej pamięci i nie umożliwia użytkownikowi ręcznej konfiguracji przydzielanych zasobów.



Rys. 6.2. Mediana wykorzystania pamięci dla 5 uruchomień programu, z jednym egzekutorem dla wersji Spark YARN

6.1.2. Klaster Apache Spark

Program do zliczania słów został również zaimplementowany i uruchomiony na klastrze Apache Spark. Na początku na wejście podano 418 książek zapisanych w oddzielnych plikach. Tabela 6.2. przedstawia medianę wyników dla 10 uruchomień. Niestety, monitoring klastra Sparka nie pokazuje całkowitego zużycia pamięci przez aplikację. Można jedynie na podstawie konfiguracji

wyliczyć minimalną potrzebną wartość, ale taka liczba nie jest wystarczająca do porównywania rzeczywistej wydajności.

W tabeli 6.2. widzimy, że wyniki poszczególnych czasów są bardzo podobne do tych z tabeli 6.1. – czyli Spark bez żadnego wcześniejszego przygotowania plików przetwarza je tak samo dobrze, jak Hadoop, który wymagał połączenia ich w jeden plik.

Całkowity czas	46 s
Czas wykonania programu	41,74 s
Czas mapowania	19 s
Czas redukowania + zapisu do pliku wynikowego	22 s
Czas działania *garbage collectora*	5 s

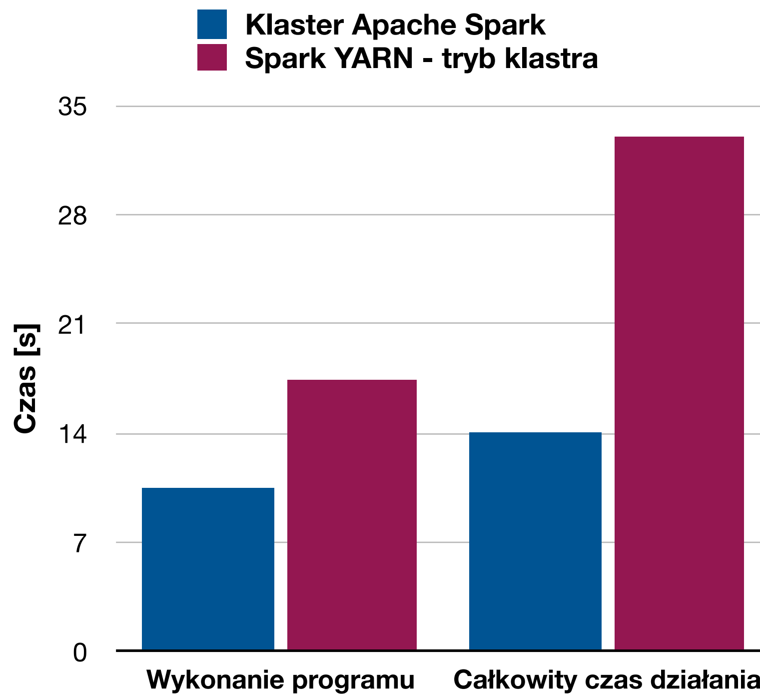
Tabela 6.2. Mediana dla 10 uruchomień programu, z danymi w oddzielnych plikach

Wyniki zostały zapisane do pliku tekstowego, ale w tym przypadku słowa są nieposortowane w żaden sposób, trudniej więc przeglądać go przez użytkownika. Po dodaniu do programu funkcji sortowania wyników, całkowity czas to już 66 s. Operacja ta jest dość kosztowna, gdyż wymaga ponownego przejścia przez wszystkie wiersze, tak jak w mapowaniu. Znacznie lepsze wyniki osiągnięte zostały po połączeniu plików wejściowych w jeden – zliczanie słów z sortowaniem zajęło 16 s (mediana dla 5 uruchomień), a bez sortowania 14 s, co pokazuje tabela 6.3. Widzimy, że główną zaletą jest znacznie krótszy czas mapowania niż w przypadku programu napisanego w Hadoop MapReduce.

Całkowity czas	14 s
Czas wykonania programu	10,55 s
Czas mapowania	7 s
Czas redukowania + zapisu do pliku wynikowego	2 s
Czas działania *garbage collectora*	0,8 s

Tabela 6.3. Mediana dla 5 uruchomień programu, z danymi jednym pliku, bez sortowania

Powyższe wyniki można również porównać z wynikami uruchomień programu Sparka na klastrze Hadoopa, gdyż jest to identyczny kod. W obu przypadkach porównywane są uruchomienia programu z jednym egzekutorem, z tym, że dla własnego klastra Sparka egzekutorem jest sterownik. Wykres 6.3. ilustruje, że różnice widoczne są nawet w samym czasie wykonania programu, co pokazuje, że dla małych rozmiarów danych klaster Sparka jest szybszy niż klaster Hadoopa.



Rys. 6.3. Mediana dla 5 uruchomień programu, z jednym egzekutorem, dla własnego klastra Sparka i dla programu zaimplementowanego w Sparku, ale uruchomionego przy użyciu Hadoop YARN

Analizowany program korzysta ze struktury RDD, która została przedstawiona w listingu 4. Widzimy tu fragment kodu odpowiadającego za operacje na pliku tekstowym w typowym stylu mapowania i redukowania.

```
val counts: RDD[(String, Int)] = textFile
  .flatMap(line => line.split(regex))
  .filter(_.length != 0)
  .map(word => (word.toLowerCase, 1))
  .reduceByKey(_ + _)
counts.saveAsTextFile(outPath)
```

Listing 4. Zliczanie słów w pliku – implementacja korzystająca ze struktury RDD

Nowsze wersje Sparka, w tym wersja używana w niniejszej pracy, posiadają strukturę DataFrame, stworzoną jako lepiej działającą alternatywę do RDD. Listing 5. przedstawia fragment implementacji programu wykorzystujący DataFrame. Widzimy, że choć działanie i wynik operacji jest identyczny, jak w przypadku listingu 4, to używane są różne funkcje. DataFrame wymaga podziału na kolumny, a funkcje `select()`, czy `groupBy()` są stworzone na wzór operacji w języku SQL. Czas wykonania takiego programu dla danych połączonych w jeden plik

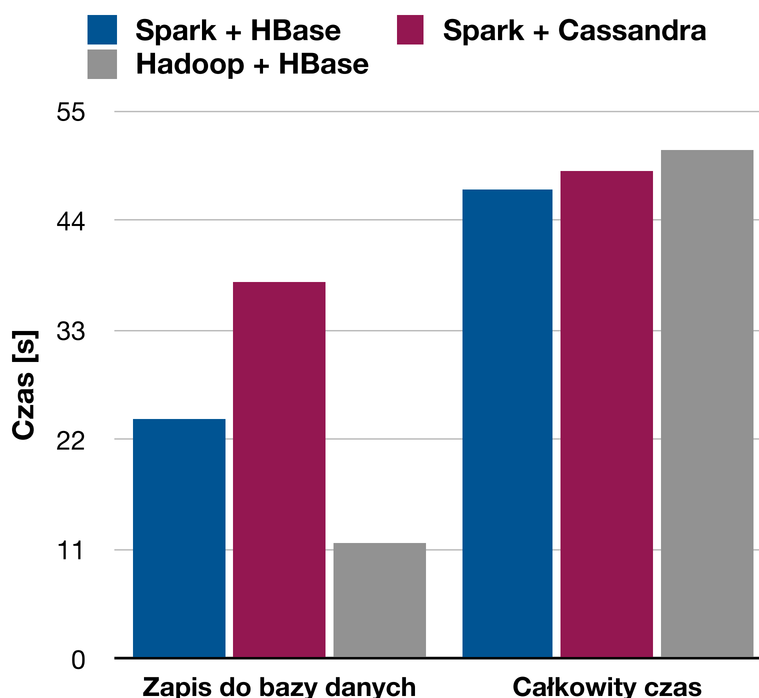
wejściowy to 18,02 s (mediana z 5 uruchomień), podczas gdy dla RDD było to 10,55 s. Widać więc, że mając dane bez określonego schematu, lepiej sprawdza się prostsza struktura RDD.

```
val counts: DataFrame = textFile
  .withColumn("word", explode(split(regex)))
  .select(lower(col("word")))
  .filter(not(col("word") === lit("")))
  .groupBy("word").count
```

Listing 5. Zliczanie słów w pliku – implementacja korzystająca ze struktury DataFrame

6.1.3. Bazy danych

Zapisywanie wyników do zwykłego pliku tekstowego może się sprawdzać w niektórych przypadkach, ale zazwyczaj nie jest to najlepszy sposób do przechowywania danych, bo trudno je analizować i przeszukiwać. Dlatego frameworki do przetwarzania danych łączy się z bazami danych. W skład ekosystemu Hadoop wchodzi baza HBase, która została stworzona z myślą o zapisie i wczytywaniu dużych ilości danych, więc używanie jej w aplikacjach MapReduce jest naturalnym wyborem. W tym podrozdziale przedstawione zostanie porównanie HBase z Cassandrą – obie bazy mają podobne zastosowanie i sposób działania.



Rys. 6.4. Mediana dla 5 uruchomień programu, dla różnych klastrów i baz danych

Wykres 6.4. przedstawia porównanie czasu działania programu do zliczania słów dla trzech przypadków: program napisany w Sparku i uruchomiony na klastrze Hadoopa z bazą HBase (w tym przypadku wybrana została najlepsza konfiguracja z wcześniej analizowanych: tryb klastra z jednym egzekutorem) oraz ten sam program uruchomiony na własnym klastrze Sparka z bazą Cassandra, a także program zaimplementowany przy użyciu Hadoop MapReduce z zapisem do bazy HBase. Na wykresie przedstawiony jest całkowity czas działania zadania oraz czas samego zapisu do bazy danych. Widzimy, że całkowity czas działania jest bardzo podobny we wszystkich przypadkach, ale za to HBase wypada zdecydowanie korzystniej pod względem czasu zapisu danych. Baza HBase została początkowo stworzona specjalnie do programów zaimplementowanych w Hadoop MapReduce i poniższe wyniki potwierdzają, że właśnie w tym połączeniu sprawdza się najlepiej pod względem szybkości zapisu. Widzimy też kolejny raz, że klaster Sparka jest ogólnie szybszy, ale połączenie go z Cassandrą w tym przypadku się nie sprawdziło.

6.2 Przetwarzanie danych uporządkowanych

Dane uporządkowane to takie, które są zbudowane według określonego schematu – mogą to być zarówno pliki CSV, JSON, XML, dane z wyszukiwarek internetowych, czy rekordy z baz danych kolumnowych. Takie dane już bezpośrednio nadają się do analizy i agregacji, a ich przetwarzanie polega zwykle na zapisaniu ich w nowym modelu danych, dostosowanym do innych potrzeb.

Jako przykład stworzony został program do sprawdzania, czy e-mail użytkownika lub jego hasło są zagrożone w wyniku naruszenia danych. Idea działania została oparta na serwisie „Have I Been Pwned?” [40]. Wygenerowane zostały dane symulujące rzeczywiste dane użytkowników, które mogłyby się znajdować w bazach różnych sklepów czy serwisów internetowych. Każdy plik z danymi zapisany został w formacie CSV i posiada 9 kolumn: imię i nazwisko, adres e-mail, numer telefonu, hasło, numer karty kredytowej, adres, miasto, kod pocztowy i kraj. Założono, że są to dane, które wyciekły z 49 sklepów internetowych.

Pierwszym etapem działania programu jest wyodrębnienie i zapis jedynie potrzebnych informacji – wszystkie dane nie powinny się znaleźć w ogólnodostępnej bazie, gdyż zachęciłoby to hakerów do włamywania się do niej. Użytkownik potrzebuje jedynie wiedzieć, czy jego e-mail lub hasło znalazły się wśród danych, które wyciekły, i w jakiej bazie były one pierwotnie zapisane. Ponadto adresy e-mail i hasła zapisywane są w oddzielnych plikach lub tabelach dla dodatkowego zabezpieczenia. Druga część programu jest interaktywna dla użytkownika – może on podać swój adres e-mail lub hasło i otrzyma informację, czy znajdują się one w jednej z baz danych, a jeśli tak, to jakie to są bazy.

Zbiór danych wykorzystywanych w tym przykładzie liczy 2 miliony wierszy, zapisanych łącznie w 20 000 plików. Łączny rozmiar danych to 331 MB.

6.2.1. Spark DataFrame

Pierwsza implementacja programu została napisana w Sparku, przy użyciu struktury DataFrame. Jest to najlepszy wybór dla danych o określonym schemacie, gdyż kolumny z pliku CSV są przenoszone bezpośrednio do kolumn będących częścią DataFrame. Wszelkie operacje na kolumnach są czytelne i proste do przeprowadzenia, co jest dużą zaletą, szczególnie w przypadkach, gdy konieczne jest przeprowadzenie wielu transformacji i agregacji.

Wykorzystywane w tym przykładzie dane zapisane są w 49 folderach, odpowiadających 49 bazom użytkowników z różnych witryn internetowych. W plikach CSV nie ma informacji o tym, skąd te dane pochodzą, więc jeśli wszystkie pliki wczytane zostaną naraz do jednego DataFrame'a, to stracimy potrzebne informacje. Konieczne jest więc dodawanie kolumny z nazwą folderu (czyli nazwą bazy) w trakcie odczytu plików.

Listing 6. przedstawia implementację odczytu plików CSV, gdzie `websites` jest listą nazw wszystkich folderów z danej ścieżki. Każdy plik wczytywany jest z pominięciem nagłówka z nazwami kolumn, a także z zastosowaniem przygotowanego wcześniej schematu, będącego strukturą określającą nazwy i typy kolumn, które zapisujemy do DataFrame'a. Odczytywanie plików w ten sposób skutkuje stworzeniem tablicy `Array[DataFrame]`, o długości odpowiadającej liczbie folderów, czyli w tym przypadku 49. Jest to nieefektywne dla dalszych operacji, więc zastosowana została redukcja przy użyciu funkcji `union`, która łączy wszystkie elementy w jeden DataFrame.

```
val df: DataFrame = websites.map(directory =>
  spark.read
    .option("header", value = true)
    .schema(schema)
    .csv(s"$path/$directory/*.csv")
    .withColumn("database_name", lit(directory)))
  .reduce(_ union _)
  .coalesce(4)
```

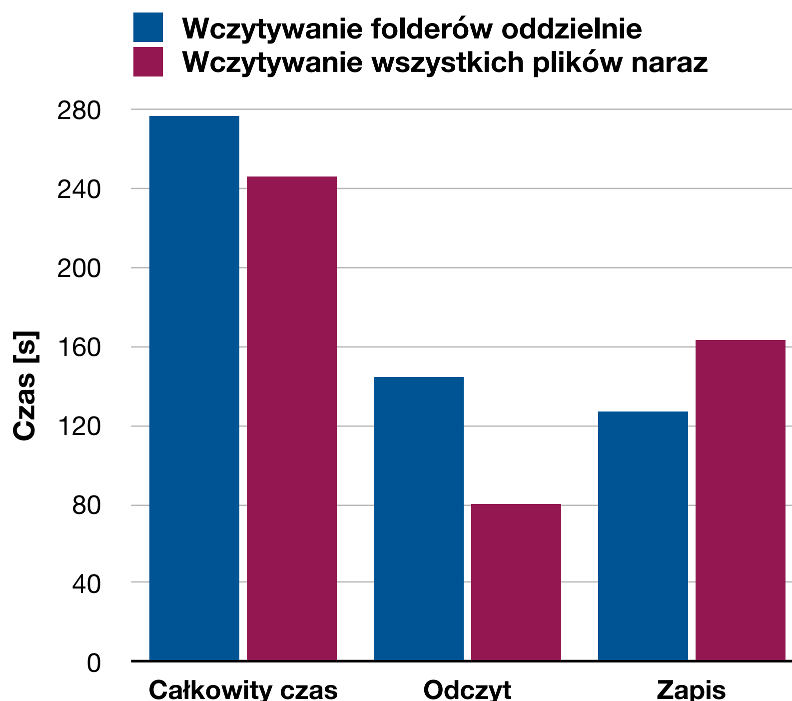
Listing 6. Odczyt plików CSV do struktury DataFrame, z każdego folderu po kolei

W przykładzie ze zliczaniem słów klaster uruchomiony został z tylko jednym egzekutorem, gdyż rozmiar danych był zbyt mały, by opłacało się wprowadzać jakąkolwiek równoległość. W tym przypadku zwiększenie liczby egzekutorów przynosi już korzyść – czas działania programu jest krótszy. Program jest uruchamiany na jednej maszynie z dwurdzeniowym procesorem, więc liczbę egzekutorów można jedynie zwiększyć do dwóch. Przetwarzane dane zostały podzielone na cztery równe części za pomocą funkcji `coalesce` (listing 6.), według rekomendacji w dokumentacji Apache Spark, w której zalecane są 2-3 zadania na rdzeń klastra [41]. Każda partycja danych odpowiada jednemu zadaniu, więc przy podziale na cztery części mamy po dwa zadania na rdzeń.

Program podzielono na dwa etapy: pierwszy z nich to odczyt danych z pliku CSV do struktury DataFrame, natomiast drugi etap to wybór odpowiednich kolumn i zapisanie ich do dwóch oddzielnych plików CSV – jeden z adresami e-mail, a drugi z hasłami. Całkowity czas działania wyniósł 4,6 min (mediana dla 5 uruchomień), czas pierwszego etapu (odczytu) to 2,4 min, natomiast czas zapisu do dwóch plików to 2,1 min. Analizując metryki uruchomionej aplikacji zauważono, że podczas odczytu plików Spark tworzy 49 oddzielnych zadań, co wyraźnie wydłuża działanie programu. Chcąc naprawić ten problem, zaimplementowane zostało ulepszenie zaprezentowane w listingu 7. Widzimy, że tym razem wszystkie pliki ze ścieżki wczytywane są naraz, do jednego DataFrame'a. Nazwa folderu wyodrębniona zostaje z wartości funkcji `input_file_name()`, która zwraca pełną ścieżkę pliku, z którego pochodzą wczytywane dane

```
val df: DataFrame = spark.read
  .option("header", value = true)
  .schema(schema)
  .csv(s"$path/*/*.csv")
  .withColumn("database_name", regexp_extract(
     input_file_name(), """([^/]*)/[^/]+\.csv$""", 1))
  .coalesce(4)
```

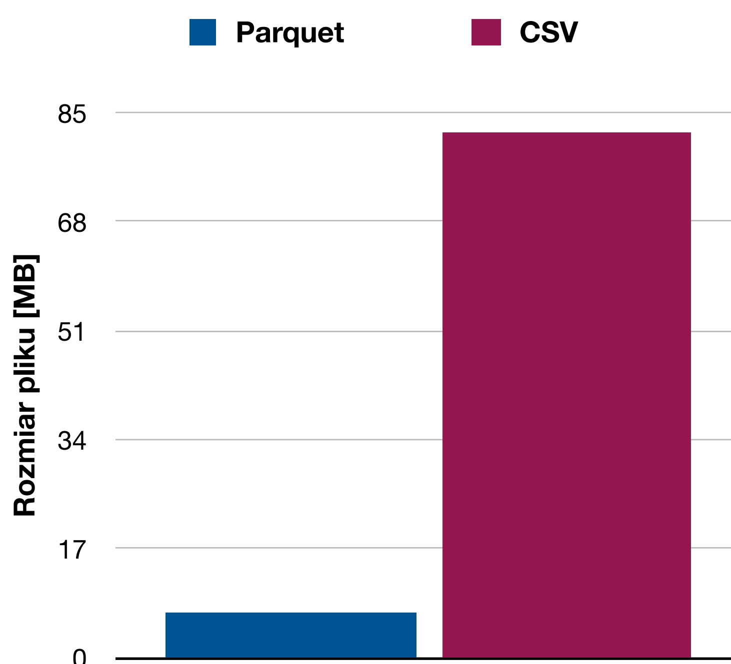
Listing 7. Odczyt plików CSV do struktury DataFrame, ze wszystkich folderów jednocześnie



Rys. 6.5. Mediana dla 5 uruchomień programu, dla różnego sposobu wczytywania plików

Modyfikacja wprowadzona w listingu 7. wpływa tylko na pierwszą część programu – kod zapisu do plików wynikowych pozostaje bez zmian. Wykres 6.5. przedstawia porównanie wyników dla uruchomień programu przy wczytywaniu folderów po kolei oraz jednocześnie. Wyniki można uznać za dość zaskakujące. Co prawda po wprowadzeniu modyfikacji czas odczytu znacznie się skrócił, za to wydłużył się czas zapisu i w efekcie całkowite czasy działania dla obu wersji programu różnią się tylko o 11%. Na pierwszy rzut oka wydłużenie czasu zapisu może się wydawać nielogiczne, bo w obu przypadkach zapisujemy jeden DataFrame w czterech partycjach, nie zmienił się też ani kod, ani dane. Należy jednak wziąć pod uwagę, że Spark korzysta z tzw. leniwej ewaluacji i operacje na strukturach RDD czy DataFrame nie wykonują się od razu w miejscu ich wywołania, ale dopiero w momencie, gdy potrzebny jest wynik tej operacji. Spark zapisuje metryki dotyczące wykonanego planu zapytań. Można więc sprawdzić, co i w jakiej kolejności zostało rzeczywiście wykonane. Okazuje się, że podczas zapisu DataFrame'a z listingu 6. nadal występuje podział na 49 części, a funkcja `union()` dodała jedynie interfejs opakowujący te partycje, dzięki czemu z perspektywy piszącego kod można operować na jednej strukturze.

Spark wspiera wiele różnych formatów plików, więc warto się zastanowić, czy CSV jest odpowiednim formatem do zapisu wyników programu. Największą zaletą pliku CSV jest jego czytelność i łatwość modyfikacji – także przez osoby niebędące programistami. Jednak jeśli chcemy po prostu przechowywać dane i korzystać z nich za pomocą programu napisanego w Sparku, można skorzystać z bardziej kompaktowych formatów. Dla danych uporządkowanych, przechowywanych w kolumnach, dobrą opcją jest format Parquet [42]. Zajmuje niewiele miejsca, gdyż dane zapisywane są w postaci binarnej, do tego umożliwia bardzo szybki i efektywny odczyt danych.



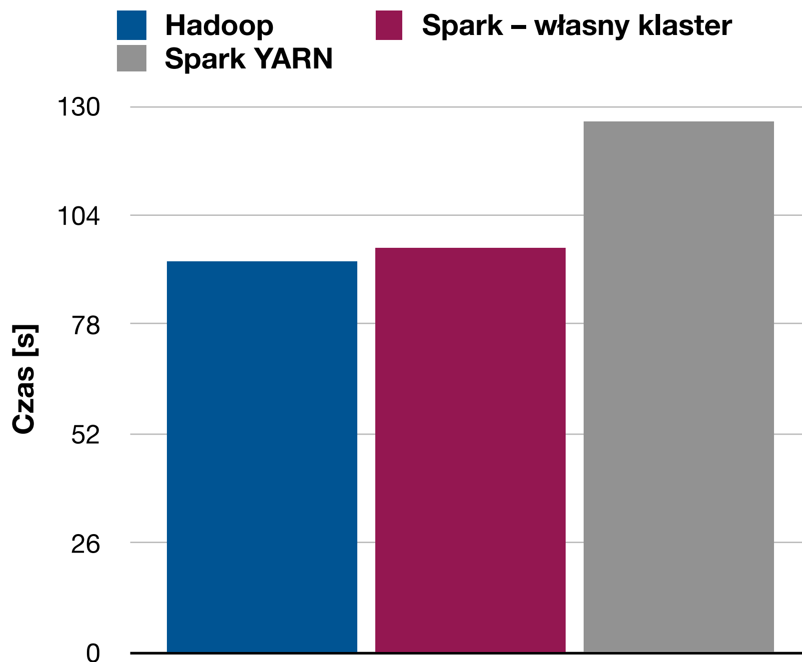
Rys. 6.6. Rozmiar pliku wynikowego w zależności od formatu zapisu

Wykres 6.6. przedstawia wielkość zapisanego pliku wynikowego w zależności od jego formatu. Widzimy, że różnica jest spora, więc już samo zapisanie pliku w innym formacie może znacząco obniżyć koszty przechowywania danych. Czas zapisu również wypada lepiej dla formatu Parquet – mediana czasu wyniosła 155 s, podczas gdy dla formatu CSV było to 163,5 s (w obu przypadkach do odczytu wykorzystana została implementacja z listingu 7.).

6.2.2. Hadoop MapReduce

Ten sam program został zaimplementowany również przy użyciu frameworku Hadoop MapReduce. Przy analizie programu ze zliczaniem słów widać było, że Hadoop nie jest dostosowany do przetwarzania większej liczby małych plików. W tym przypadku dane zapisane są do 20 000 plików, więc próba ich przetworzenia w programie korzystającym z Hadoop MapReduce doprowadziła do zawieszenia systemu. Konieczne było więc połączenie tych plików – w tym podrozdziale omawiane będzie przetwarzanie danych znajdujących się w 49 plikach CSV, przy czym każdy z nich odpowiada jednej bazie danych użytkowników. Po połączeniu plików rozmiar danych to 271 MB.

Aby móc przeprowadzić porównanie, dla tak samo połączonych danych uruchomiono również program zaimplementowany w Sparku – na jego własnym klastrze, a także korzystając z menadżera zasobów Hadoop YARN. Wykres 6.7. przedstawia całkowity czas dla poszczególnych opcji. Widzimy, że najlepiej wypadł program zaimplementowany w Hadoop MapReduce, choć jego przewaga czasowa nad Sparkiem nie jest duża, bo wynosi 3 s.



Rys. 6.7. Mediana dla 5 uruchomień programu, z danymi wejściowymi zapisanymi do 49 plików

Powyższe wyniki różnią się znacznie od tych otrzymanych podczas uruchamiania przykładu ze zliczaniem słów. Tabela 6.4. przedstawia porównanie uzyskanych czasów dla obu przykładów, wraz z wyliczoną różnicą procentową ze wzoru: $\frac{B-A}{A} \cdot 100\%$, gdzie A to wartość z przykładu ze zliczaniem słów, a B to wartość z przykładu analizowanego w tym podrozdziale. Możemy w ten sposób porównać wrażliwość sposobu implementacji na rozmiar danych. We wszystkich poniższych tabelach nazwą „Hadoop" określona jest implementacja przy użyciu Hadoop MapReduce, „Spark YARN" to implementacja korzystająca z frameworku Spark i uruchomiona na środowisku Hadoop YARN w trybie klastra, natomiast „Spark – własny klaster" to implementacja korzystająca z frameworku Spark i uruchomiona na jego wbudowanym klastrze.

Widzimy, że program korzystający z Hadoop MapReduce wypadł czasowo najgorzej w pierwszym przykładzie, ale już w drugim najlepiej – czas działania zwiększył się tylko dwukrotnie dla prawie siedmiokrotnie większego rozmiaru danych wejściowych. Co ciekawe, te różnice występują też porównując program napisany w Sparku i uruchomiony na jego własnym klastrze, z tym samym programem uruchomionym na klastrze Hadoopa. Widzimy, że nie tylko kod aplikacji wpływa na czas działania i klaster Sparka stosunkowo gorzej optymalizuje przetwarzanie większej ilości danych. Mimo to, uruchomienie programu na własnym klastrze Sparka w obu przykładach w skali bezwzględnej wypada lepiej, gdyż nadal część czasu mija na dodatkowym łączeniu się i rozdzielaniu zasobów przez YARN.

	Przykład: zliczanie słów	**Przykład: wyciek danych**	**Różnica procentowa**
Rozmiar danych wejściowych	39,3 MB	271 MB	590%
Całkowity czas: Hadoop	44,5 s	**93 s**	**109%**
Całkowity czas: Spark YARN	33 s	126 s	282%
Całkowity czas: Spark - własny klaster	**14 s**	96 s	586%

Tabela 6.4. Porównanie mediany całkowitego czasu działania dla różnych programów i sposobu implementacji

Rozmiar danych wejściowych	271 MB	542 MB	+100%
Całkowity czas: Hadoop	**93 s**	**122 s**	+31%
Całkowity czas: Spark YARN	126 s	150 s	**+19%**
Całkowity czas: Spark - własny klaster	96 s	156 s	+63%

Tabela 6.5. Porównanie mediany całkowitego czasu działania dla różnych rozmiarów danych wejściowych w przykładzie z wyciekami danych, z wyznaczoną różnicą procentową w ostatniej kolumnie

Aby móc wysnuć bardziej miarodajne wnioski, porównana została skalowalność w ramach tego samego programu. Tabela 6.5. przedstawia zestawienie całkowitego czasu działania dla dwukrotnie zwiększonego rozmiaru danych. W tym przypadku Hadoop wypada już zdecydowanie korzystniej w porównaniu z pozostałymi opcjami, dodatkowo można zauważyć, że różnica

czasowa między uruchomieniem programu Sparka na jego własnym klastrze i przy użyciu Hadoop YARN wyraźnie się zmniejszyła.

Jednym z elementów, które mają mierzalny wpływ na otrzymane wyniki jest czas działania *garbage collectora*, co przedstawione jest w tabeli 6.6. Widzimy, że połączenie Spark YARN najlepiej wypada w tej kategorii, a dwukrotne zwiększenie rozmiaru danych nie wpłynęło w żaden sposób na czas utylizacji zasobów. Oczywiście wartości rzędu kilku sekund nie mają zbyt wielkiego udziału w całkowitym czasie działania aplikacji, ale należy zwrócić uwagę na ich dynamikę wzrostu – przy terabajtach danych czas działania *garbage collectora* może już znacząco zaważyć na wydajności aplikacji.

Rozmiar danych wejściowych	271 MB	542 MB	+100%
Hadoop	**0,9 s**	1,3 s	+44%
Spark YARN	1 s s	**1 s**	**+0%**
Spark - własny klaster	2 s	4 s	+100%

Tabela 6.6. Porównanie mediany czasu działania *garbage collectora* dla różnych rozmiarów danych wejściowych, z wyznaczoną różnicą procentową w ostatniej kolumnie

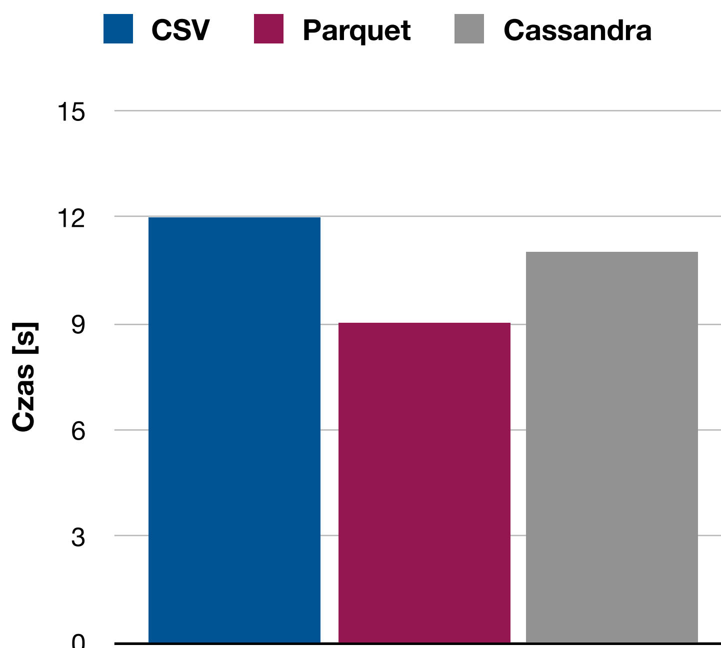
Choć czasowo Hadoop MapReduce wypada korzystniej, to pod względem łatwości implementacji i możliwości rozwoju programu Spark sprawdza się lepiej. Program zaimplementowany w tym podrozdziale jest stosunkowo prosty koncepcyjnie, ale już sprawił pewne problemy implementacyjne w Hadoop MapReduce – konieczne było stworzenie dodatkowej klasy do odczytu plików z różnych folderów, gdyż framework nie przewidywał takiej opcji. Dodatkowo Spark DataFrame zapewnia wiele funkcji do agregacji i transformacji danych, a Hadoop wymaga napisania wszystkiego w formie mapowania i redukowania, co może skutkować wieloma linijkami kodu, by osiągnąć ten sam efekt, który daje wywołanie jednej funkcji Sparka.

6.2.3. Odczyt danych

W poprzednich podrozdziałach przeanalizowana została część programu związana z zapisem danych do pliku. Druga część tego programu korzysta z zapisanych wcześniej danych i jest interaktywna dla użytkownika, który może podać e-mail lub hasło i wyświetlone zostaną nazwy baz danych, w których znajdują się te informacje. Cały program polega więc wyłącznie na odczycie gotowych danych z pliku lub bazy i przefiltrowaniu wyniku, więc można tu porównać różnice czasowe we wczytywaniu danych z różnych formatów i baz.

Program zaimplementowano przy użyciu frameworku Spark i uruchomiono na jego własnym klastrze. Wykres 6.8. przedstawia czas odczytu dla plików CSV, Parquet oraz dla danych zapisanych w bazie Cassandra. Program został uruchomiony również dla danych zapisanych do bazy

HBase, ale mediana wyniku wyniosła 40 s, więc nie został on pokazany na wykresie, by nie zaburzać skali. Parquet wypadł najlepiej podczas porównania czasu zapisu, widzimy też, że również dla odczytu dobrze się sprawdza. Jego główną wadą jest to, że aby móc mieć wgląd w dane, potrzebny jest program napisany w Sparku. Dobrą opcją więc może być użycie Cassandry, której czas odczytu wypada lepiej niż CSV, a jako baza danych wnosi wiele możliwości związanych z przeprowadzaniem zapytań i analizy danych.



Rys. 6.8. Mediana dla 5 uruchomień programu napisanego we frameworku Spark, dla różnych sposobów zapisu danych wejściowych

Problemy sprawia implementacja tego przykładu w Hadoop MapReduce – framework wymaga zdefiniowania programu o określonej strukturze, więc nawet prosty przykład z wyszukiwaniem adresu e-mail w bazie jest uruchamiany w ten sam sposób, co zadanie do przetwarzania danych. Mediana czasu odczytu uzyskana dla programu MapReduce to 26 s dla plików zapisanych w formacie CSV. Do tego nie było możliwe zastosowanie interaktywnego wprowadzania parametrów w trakcie działania programu, więc do tego typu zadań Hadoop się nie sprawdza.

6.3 Przetwarzanie danych w czasie rzeczywistym

W tej części nie zostanie przeprowadzone porównanie, gdyż Hadoop MapReduce nie ma możliwości przetwarzania danych w czasie rzeczywistym. Istnieje co prawda takie narzędzie jak Hadoop Streaming [43], jednak jego nazwa może być myląca, bo służy do uruchamiania programu MapReduce w postaci wykonywalnych skryptów napisanych w różnych językach, między innymi Python, PHP, czy R.

Spark Streaming [44] służy natomiast do strumieniowego przetwarzania danych i jest jedną z podstawowych bibliotek wchodzących w skład frameworku Spark. Jego główną cechą jest przetwarzanie danych w postaci mikro partycji (ang. *micro-batch*), czym różni się od popularnych frameworków, takich jak Apache Storm [45] czy Apache Flink [46], które zostały stworzone do procesowania danych natychmiastowo. Spark Streaming imituje przetwarzanie danych w czasie rzeczywistym wprowadzając zdyskretyzowany strumień danych DStream, będący zbiorem struktur RDD.

Zaletą Spark Streaming jest prostota, z jaką można przekształcić zwykłą aplikację napisaną w Sparku, w taką, która przetwarza dane w czasie rzeczywistym. W listingu 8. na zielono zaznaczone zostały różnice w stosunku do programu do zliczania słów analizowanego w podrozdziale 6.1, w listingu 4. Widzimy, że wystarczy jedynie stworzyć obiekt `StreamingContext`, by możliwe było wczytywanie strumienia danych z podanej ścieżki, a dalsze operacje na DStream są już takie same jak dla RDD. Aplikacja przetwarzająca dane w czasie rzeczywistym charakteryzuje się tym, że nigdy samoczynnie się nie zakończy, chyba że nastąpi przerwanie procesu ze strony programisty – zapewnia to funkcja `awaitTermination()`.

```
val ssc = new StreamingContext(sc, Seconds(1))
val textFile = ssc.textFileStream(inPath)
val counts: DStream[(String, Int)] = textFile
  .flatMap(line => line.split(regex))
  .filter(_.length != 0)
  .map(word => (word.toLowerCase, 1))
  .reduceByKey(_ + _)
counts.saveAsTextFiles(outPath)
ssc.start()
ssc.awaitTermination()
```

Listing 8. Program do zliczania słów w czasie rzeczywistym

Dla programów korzystających z danych o określonej strukturze wprowadzony został model Structured Streaming, oparty na strukturach DataFrame i Dataset. W tym przypadku przepisanie aplikacji na strumieniowe przetwarzanie danych również nie stwarza większych problemów i sprowadza się głównie do zamiany funkcji `read()` i `write()` na `readStream()` i `writeStream()`. Przydatną funkcją jest również możliwość zapisywania tzw. punktów kontrolnych (ang. *checkpoint*), dzięki czemu nawet po przerwaniu aplikacji, może ona wrócić do poprzedniego stanu procesowania.

Optymalizacja aplikacji przetwarzania strumieniowego przebiega podobnie jak dla przetwarzania wsadowego i ma często nawet większe znaczenie, gdyż użytkownikowi końcowemu zależy, by mógł otrzymać wyniki jak najszybciej.

6.4 Podsumowanie

Wyniki otrzymane w tym rozdziale potwierdzają założenia architektury lambda – mając dużo danych do przetworzenia naraz, Hadoop MapReduce wypada lepiej pod względem czasu, więc użycie go do obsługi warstwy wsadowej jest dobrym pomysłem. Aby mieć możliwość przetwarzania danych w czasie rzeczywistym, dodaje się warstwę strumieniową, którą obsługuje Spark Streaming. Problemem takiego układu jest jednak to, że wymaga on dwóch różnych od siebie implementacji – jednej w Javie, przy użyciu bibliotek MapReduce, a drugiej w Scali (lub Pythonie), korzystając z bibliotek Spark. Trudności sprawia synchronizacja tak napisanych programów – jeśli wprowadzimy zmianę w jednym z nich, musimy się zastanawiać, jak wprowadzić analogiczną zmianę w drugim i czy jest ona w ogóle możliwa. Dlatego alternatywą w warstwie wsadowej może być program napisany w Sparku, ale uruchomiony na klastrze Hadoopa – wyniki z podrozdziału 6.2. pokazały, że ma lepszą skalowalność niż własny klaster Sparka. Dzięki temu wszystkie aplikacje są spójne między sobą, bo korzystają z tej samej technologii, do tego różnice implementacyjne między zwykłym programem napisanym w Sparku a tym korzystającym z bibliotek Spark Streaming czy Structured Streaming nie są duże, co pozwala na dzielenie części kodu między nimi.

Architektura lambda korzystająca z jednej technologii umożliwia uproszczenie do postaci architektury kappa. Jeśli nie ma potrzeby przetwarzania naraz bardzo dużej ilości danych, to optymalną opcją jest w tym przypadku przejście w całości na przetwarzanie strumieniowe – daje to wiele zalet w postaci natychmiastowych wyników analizy. Dodatkowo, projektując system w architekturze kappa, nie trzeba się martwić nadmiarem technologii, bo Spark zapewnia biblioteki zarówno do przetwarzania danych, jak i ich analizy i uczenia maszynowego.

Uniwersalność jest główną przewagą frameworku Spark nad Hadoop MapReduce. Konieczność implementacji programu jedynie w postaci zadań mapujących i redukujących znacznie ogranicza możliwości programisty. Dodatkowo w Scali i Sparku można napisać bardziej zwięzłe programy niż w Javie i MapReduce. Przeprowadzone zostało porównanie liczby linijek kodu we wszystkich programach zaimplementowanych w tej pracy i programy Hadoop MapReduce mają średnio o 70% więcej kodu niż te napisane w Sparku.

W skład ekosystemu Hadoop wchodzi wiele narzędzi, które łącznie pozwalają na stworzenie uniwersalnego i wielozadaniowego systemu. Jednak zwykle prościej jest używać różnych bibliotek w ramach tej samej technologii, co może być przyczyną, dla której Hadoop zaczyna być uznawany za technologię przestarzałą. Mimo to jest obecny na wielu platformach chmurowych i w następnym rozdziale przedstawione zostanie działanie obu technologii w nowoczesnym ujęciu.

7. Architektura w chmurze Azure

W tym rozdziale prezentowane będzie działanie programów uruchomionych na chmurze Azure (parametry wymienione zostały na początku rozdziału 5.).

Usługa HDInsight została uruchomiona w wersji 3.6, wykorzystując technologie Apache Hadoop w wersji 2.7. oraz Apache Hive w wersji 2.0. Klaster Databricks został uruchomiony w wersji 6.5. z Apache Spark w wersji 2.4.5.

7.1 Przetwarzanie danych

W tej części uruchamiany będzie program do przetwarzania rzeczywistych danych pewnego sklepu internetowego, pobranych ze strony Kaggle [47]. Dane zapisane są w formacie CSV i posiadają informacje o produktach, które były przeglądane lub kupione przez klienta. Celem analizowanego przykładu jest wyznaczenie łącznej ceny dla każdej marki i kategorii produktu, by mieć wgląd w to, jakie produkty mają najwyższą cenę, a przy tym cieszą się najwyższym zainteresowaniem. Dane zostały zapisane w magazynie Azure Blob Storage, z którego mogą korzystać jednocześnie klastry HDInsight oraz Databricks.

Rozmiar analizowanego zbioru danych to 9 GB i został podzielony na pliki w dwóch konfiguracjach:

- 136 plików, po około 67 MB każdy;
- 7 plików, z czego rozmiar sześciu to około 1,33 GB, a siódmego – 1 GB.

Zbadane zostało także przetwarzanie większej liczby bardzo małych plików – rozmiar zbioru to 621 KB, a dane zostały podzielone na 4720 plików, po około 135 B każdy.

7.1.1. HDInsight

Analizowany przykład został celowo stworzony, by jak najlepiej wpisać się w operacje mapowania i redukowania, więc implementacja w Hadoop MapReduce nie sprawiała problemów. Początkowo kod zawierał nadmiarowe zmienne, w których zapisywane były wszystkie kolumny z pliku wejściowego, choć potrzebne były tylko trzy z nich. Tak napisany program porównany został w pięciu próbach z implementacją nie zawierającą niewykorzystywanych zmiennych. Wyniki

przedstawione zostały w tabeli 7.1. Czas działania *garbage collectora* podawany jest w statystykach jako suma dla wszystkich równolegle działających egzekutorów, stąd wartość może być często wyższa niż całkowity czas wykonywania zadania. Za każdym razem wykonanych zostało 136 zadań mapujących (po jednym zadaniu na każdy plik) oraz 4 zadania redukujące (wartość została ustawiona w implementacji programu jako optymalna w tym przypadku). Nadmiarowe zmienne zostały zainicjalizowane w funkcji mapującej, stąd duże różnice w średnim czasie mapowania widocznym w tabeli 7.1. Mimo że te zmienne nie wykonują żadnego zadania i nie są wykorzystywane, to muszą być zutylizowane przez *garbage collector*, co sumarycznie zajmuje bardzo dużo czasu. Widać więc, jak duży wpływ na działanie aplikacji ma korzystanie z jak najmniejszej liczby zmiennych i ponowne wykorzystywanie już zainicjalizowanych, zamiast tworzenia nowych. W dalszej części podrozdziału będzie więc już rozpatrywany program z optymalną liczbą zmiennych.

	Nadmiarowe zmienne	**Brak nadmiarowych zmiennych**
Całkowity czas	7 min 47 s	1 min 57 s
Średni czas mapowania	1 min 24 s	18 s
Średni czas redukowania	6 s	6 s
Czas działania *garbage collectora*	126 min 15 s	1 min 13 s

Tabela 7.1. Mediana dla 5 uruchomień programu, porównanie dwóch implementacji

Hadoop przetwarza każdy plik w oddzielnym bloku danych, co opisane zostało już w rozdziale 6.1.1. Gdy plik ma większy rozmiar, zostaje podzielony między kilka bloków. Mając więc ustawiony rozmiar bloku danych na 64 MB i pliki o rozmiarach około 67 MB, potrzebne są dwa bloki, do których wysyłane są oddzielne żądania. Można ustawić rozmiar bloku na 67 MB, wtedy będzie wysyłane tylko jedno żądanie, by odczytać jeden plik, co skróci czas działania programu. W praktyce jednak, możemy mieć pliki o bardzo różnych rozmiarach, nie da się więc do nich dostosować optymalnego rozmiaru bloku danych. Ręczne łączenie plików to doraźne i niezbyt wygodne rozwiązanie, dlatego warto dokonać zmian w kodzie programu, by sam łączył pliki wejściowe. Biblioteka Hadoop MapReduce posiada abstrakcyjną klasę `CombineFileInputFormat`, która umożliwia implementację własnego typu, łączącego pliki wejściowe w bloki o podanym rozmiarze. Jeden blok może zawierać kilka plików lub ich fragmentów, dzięki czemu miejsce w każdym bloku danych jest w pełni wykorzystywane.

Po zaimplementowaniu łączenia plików w bloki o wielkości 64 MB, całkowity czas wykonania programu (mediana z 5 uruchomień) wyniósł 1 min 25 s, czyli przyniosło to widoczną korzyść w porównaniu z domyślnym wczytywaniem plików. Sprawdzony też został wpływ rozmiaru bloku danych na wyniki – zmieniony on został w klastrze oraz w klasie łączącej pliki na 128 MB. Otrzymane wyniki, wraz z porównaniem z czasami dla bloku o rozmiarze 64 MB, przedstawione zostały w tabeli 7.2. Widzimy, że zwiększenie rozmiaru bloku danych wpływa pozytywnie na

wynik i jest on krótszy o 7 s. Średni czas mapowania zwiększył się, ale zadań mapujących jest mniej i w efekcie mniej czasu jest poświęcane na działanie *garbage collectora*. Widzimy też, że czas redukowania pozostaje stały dla wszystkich prób przeprowadzanych w tym rozdziale, co jest spodziewanym wynikiem – zmiany dokonywane są tylko na poziomie funkcji `map()`, której wykonanie zajmuje najwięcej czasu w programie.

	Blok danych 64 MB	**Blok danych 128 MB**
Całkowity czas	1 min 25 s	1 min 18 s
Średni czas mapowania	22 s	27 s
Średni czas redukowania	6 s	6 s
Czas działania *garbage collectora*	51,6 s	42,3 s

Tabela 7.2. Mediana dla 5 uruchomień programu, dla różnych rozmiarów bloku pamięci

Ten sam przykład uruchomiono dla danych podzielonych na 7 dużych plików – wykorzystana została najlepsza konfiguracja z analizowanych powyżej, z blokiem danych o wielkości 128 MB. Otrzymane wyniki przedstawia tabela 7.3. Widzimy, że całkowity czas działania programu jest dłuższy niż ten z tabeli 7.2. Różnice występują szczególnie w przypadku czasu mapowania, gdyż mając 7 plików, klaster wykonuje tylko 7 zadań mapujących.

Całkowity czas	1 min 37 s
Średni czas mapowania	1 min 8 s
Średni czas redukowania	6 s
Czas działania *garbage collectora*	16,1 s

Tabela 7.3. Mediana dla 5 uruchomień programu, dla danych w 7 dużych plikach

Program uruchomiono również dla 4720 małych plików. Najpierw, dla porównania, nie użyto formatu `CombineFileInputFormat`. Jak można było przewidzieć, czas wykonania był bardzo długi i wyniósł 36 min 1 s (wykonana została tylko jedna próba). Po dołożeniu łączenia plików, program wykonał się w 1 min 53 s (mediana z 5 prób). Należy wspomnieć, że za każdym razem wykonywane były 4 zadania redukujące, zgodnie z ustawieniami z poprzednich przykładów. Dla tak małego rozmiaru danych niekoniecznie opłacalne jest dzielenie redukowania pomiędzy różne węzły – sprawdzono więc, jaki będzie miało wpływ zmniejszenie liczby zadań redukujących do jednego. Otrzymany czas to 1 min 33 s, czyli o 20 s krócej, niż dla programu z 4 zadaniami redukującymi. Można zatem wywnioskować, że rozdzielanie obliczeń i danych między węzłami ma swój koszt i ważne jest, by stosować zrównoleglanie odpowiednie do zadania, które jest wykonywane.

Uruchamiając usługę HDInsight z klastrem Hadoopa, mamy również możliwość korzystania z bazy Apache Hive. Aby zapisać dane do bazy, nie jest konieczne korzystanie z żadnej napisanej wcześniej aplikacji. Wystarczy stworzyć odpowiednią tabelę korzystając z języka SQL, a następnie przenieść dane do odpowiedniego folderu w magazynie Azure Blob Storage. Taka tabela pozwala na wykonywanie zapytań SQL i pobieranie wyników w postaci plików CSV. Analizowany w tym podrozdziale przykład jest możliwy do wykonania w języku SQL przy użyciu poleceń `SUM` i `GROUP BY`. Takie zapytanie wykonane zostało dla analizowanego zbioru danych składającego się z 136 plików. Mediana czasu działania zapytania dla 5 uruchomień wyniosła 45,73 s, czyli krócej niż najlepszy czas uzyskany podczas uruchamiania programu Hadoop MapReduce. Widzimy więc, że nieraz optymalizacją przetwarzania danych jest używanie odpowiednich narzędzi do odpowiednich zadań i dla prostych składniowo zapytań lepiej wypada korzystanie bezpośrednio z bazy danych i języka SQL.

7.1.2. Databricks

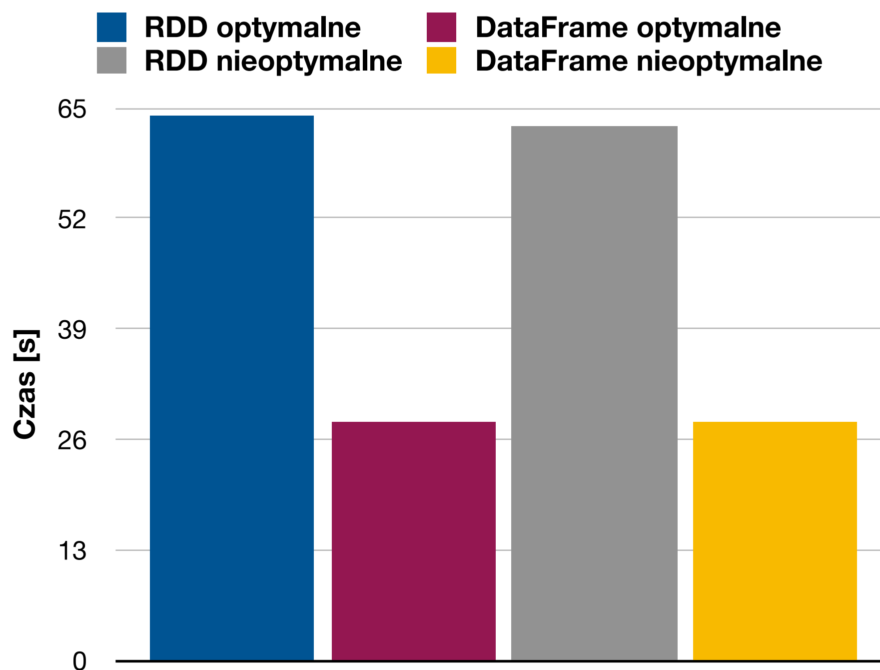
Ten sam przykład uruchomiony został na klastrze Databricks. Na początek sprawdzono działanie dla danych zapisanych w 136 plikach. Tym razem program był od początku zaimplementowany z optymalną liczbą zmiennych. Mediana całkowitego czasu działania otrzymana z 5 prób wyniosła 64 s, przy czym można zauważyć wpływ pamięci podręcznej klastra, opisany szerzej w rozdziale 5.3. Pierwsze uruchomienie programu trwało 80 s, a każde następne miało już niemal stałą, niższą wartość. Dane wejściowe załadowane zostały z magazynu Azure Blob Storage, który jest dobrą opcją do współdzielenia plików między różnymi usługami. Jeśli jednak korzystamy głównie z Databricks, to warto skorzystać z jego natywnej opcji przechowywania – Databricks File System [48] (DBFS). Jest to abstrakcja zbudowana na magazynie Azure Blob Storage, w której można tworzyć pliki, skrypty oraz tabele Apache Hive.

Interfejs Databricks kładzie główny nacisk na tworzenie i przechowywanie danych w tabelach, dlatego ten właśnie sposób został przeanalizowany. Hive jest bazą dokumentową, wspierającą różne formaty plików, więc dane zapisane w postaci CSV wczytano w środowisku Databricks i zapisano do tabeli Hive w postaci plików Parquet. Wielkość danych z 9 GB zmniejszyła się do 137,4 MB.

Przykład wczytujący dane z magazynu używał struktury RDD i traktował pliki CSV jako tekstowe. W przypadku wczytywania danych z tabeli nie można już w prosty sposób użyć RDD, napisano więc nową implementację korzystającą z DataFrame. Mediana z 5 uruchomień takiego przykładu, z odczytem i zapisem do tabeli w DBFS wyniosła 28 s, przy czym pierwsze uruchomienie trwało 62 s. Widać więc, że zmiana sposobu przechowywania wpłynęła pozytywnie nie tylko na czas działania programu, ale także na wykorzystanie pamięci podręcznej – różnica między pierwszym a następnym uruchomieniem wyniosła aż 55%.

Sprawdzony został również przypadek, gdy zostały zainicjalizowane nadmiarowe zmienne. Do obu implementacji, zarówno tej wykorzystującej RDD, jak i tej z DataFrame, dołożono

nadmiarowe zmienne, analogiczne do tych, które znalazły się w programie Hadoop MapReduce z poprzedniego podrozdziału. Wykres 7.1. przedstawia porównanie poszczególnych implementacji, gdzie „RDD" jest wersją programu z wczytywaniem plików CSV do magazynu, natomiast „DataFrame", to wersja korzystająca z tabeli w DBFS.

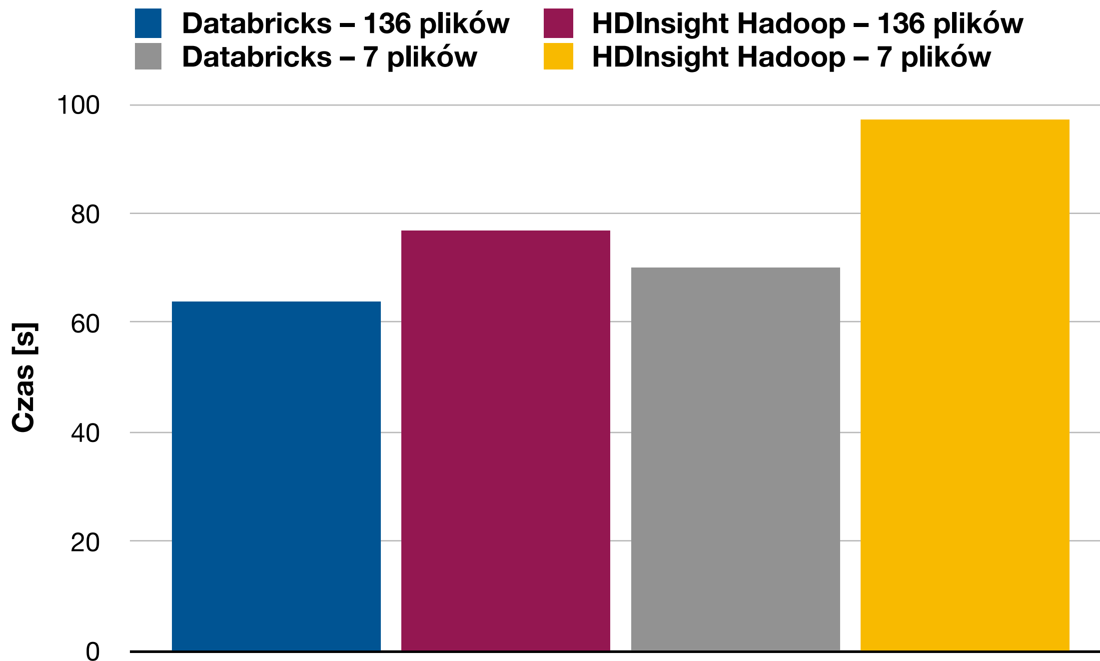


Rys. 7.1. Mediana dla 5 uruchomień programu, dla implementacji optymalnych i nieoptymalnych (zawierających nadmiarowe zmienne)

Na wykresie porównywana jest mediana czasu, ale również dla pierwszych uruchomień nie ma różnic w czasie działania pomiędzy optymalnymi i nieoptymalnymi implementacjami. Nadmiarowe zmienne nie mają żadnego wpływu na czas działania programu, z powodu leniwej ewaluacji Sparka, opisanej już w rozdziale 6.2.1. Możemy dodawać do programu wiele nowych struktur DataFrame i RDD i wykonywać na nich operacje, ale zostaną one wykonane dopiero, gdy zostanie zażądany wynik tych operacji. Zatem nadmiarowe zmienne są ignorowane przez Spark – miałyby wpływ na czas działania programu dopiero, gdy zostałaby na nich wykonana na przykład funkcja `count()` (czyli liczenie wierszy), `collect()` (zapisanie wierszy do listy) czy zapis do pliku. Widać więc dużą różnicę w stosunku do Hadoop MapReduce, który wszystkie zmienne zapisuje w pamięci, nawet jeśli nie były ani razu użyte.

Sprawdzone zostało również działanie programu dla danych zapisanych w 7 dużych plikach CSV, wczytanych do RDD. W tym wypadku mediana czasu działania wyniosła 70 s (pierwsze uruchomienie trwało 90 s). Porównanie czasów działania dla małych i dużych plików dla obu klastrów przedstawione jest na wykresie 7.2. (dla HDInsight wybrana została najlepsza implementacja, z łączeniem danych i blokiem o rozmiarze 128 MB). Widzimy, że klaster Databricks osiągnął najlepszy czas dla obu sposobów podziału danych, do tego jest mniej wrażliwy na zmianę

rozmiarów plików niż klaster Hadoop w HDInsight. Dodatkową zaletą Databricks jest również to, że dane w łatwy sposób można zapisać do tabeli – wtedy już pierwotne wielkości plików nie mają znaczenia, a czas odczytu i zapisu jest o wiele szybszy.



Rys. 7.2. Mediana dla 5 uruchomień programu, dla różnych implementacji i różnego podziału zbioru danych, w wersji bez nadmiarowych zmiennych

Na koniec sprawdzono, jak zachowa się program, podając na wejście 4720 małych plików. Również w tym przypadku wyniki okazały się znacznie lepsze, niż te otrzymane w HDInsight. Mediana czasu działania to 28 s (pierwsze uruchomienie trwało 49 s), natomiast dla plików zapisanych w tabeli w DBFS mediana czasu działania wyniosła już 17 s (42 s przy pierwszym uruchomieniu).

7.2 Wykorzystanie zasobów

W tej części przeanalizowano różnice w wykorzystaniu zasobów w klastrach HDInsight i Databricks. Dla przypomnienia – oba klastry mają te same parametry pod względem pamięci RAM, liczby rdzeni i rodzaju procesora. Rozmiar bloku danych w HDInsight ustawiony został na 128 MB. Dane wejściowe zapisano w Azure Blob Storage, skąd były odczytywane przez oba klastry. Scenariusz testowy polegał na uruchomieniu jednocześnie na klastrze dwóch programów:

- Wyznaczanie liczby π dla 16 równolegle działających funkcji mapujących oraz dla liczby iteracji $n = 2\,147\,483\,647$, będącej maksymalną wartością typu danych `Integer`, a więc największą wartością możliwą do użycia w programie. Przykład ten obciąża zarówno pamięć

(dokonujemy licznego przypisywania wartości do zmiennych), jak i procesor (obliczenia są wykonywane w „dużej" pętli).

- Przetwarzanie danych sklepu internetowego, opisane w poprzednim podrozdziale. Tym razem zbiór danych to: 136 plików po około 67 MB każdy oraz 7 plików o rozmiarach około 1,33 GB – czyli łączny rozmiar zbioru danych to około 18 GB. Program ten jest głównie obciążający dla pamięci, gdyż tworzone są nowe zmienne i przypisania, oraz dla układu wejścia-wyjścia, bo pobierane są pliki wejściowe, a wyniki zapisywane są do plików wyjściowych do magazynu danych. Aby wprowadzić również dodatkowe obciążenie dla procesora, do każdego programu przed funkcją mapującą dołożono dwie pętle: jedna wyznaczająca sumę liczb całkowitych od 1 do 10 000, a druga wyznaczająca sumę od 1 do 20 000. Aby sumy te nie został pominięte przez leniwą ewaluację Sparka, zapisano je również do pliku wynikowego.

Wyniki działania aplikacji dla poszczególnych klastrów zaprezentowano w tabelach 7.4. i 7.5. Widzimy, że czas działania obu programów w HDInsight jest wyraźnie dłuższy, jest też znaczna różnica czasowa pomiędzy programem do wyznaczania π a tym do przetwarzania danych. W Databricks natomiast całkowite czasy obu programów są stosunkowo podobne, choć lepszy wynik uzyskał program do przetwarzania danych. W tabelach podana została również średnia czasu działania oraz odchylenie standardowe od średniej, które pokazuje, jak bardzo zróżnicowane były wyniki w poszczególnych próbach. Widzimy, że w Databricks odchylenie jest zbliżone dla obu programów – może to świadczyć o tym, że każde z zadań dostawało odpowiedni przydział zasobów. W HDInsight widać znaczną różnicę w odchyleniu standardowym, które dla czasu wyznaczania liczby π ma małą wartość, po czym można wywnioskować, że klaster za każdym razem w podobny sposób optymalizował uruchomienie tego programu. Natomiast dla zadania przetwarzającego dane odchylenie standardowe jest znacznie wyższe – ponad 2 min, co może świadczyć o tym, że problem nie występuje jedynie w przydzielaniu zasobów, ale również w samym działaniu aplikacji.

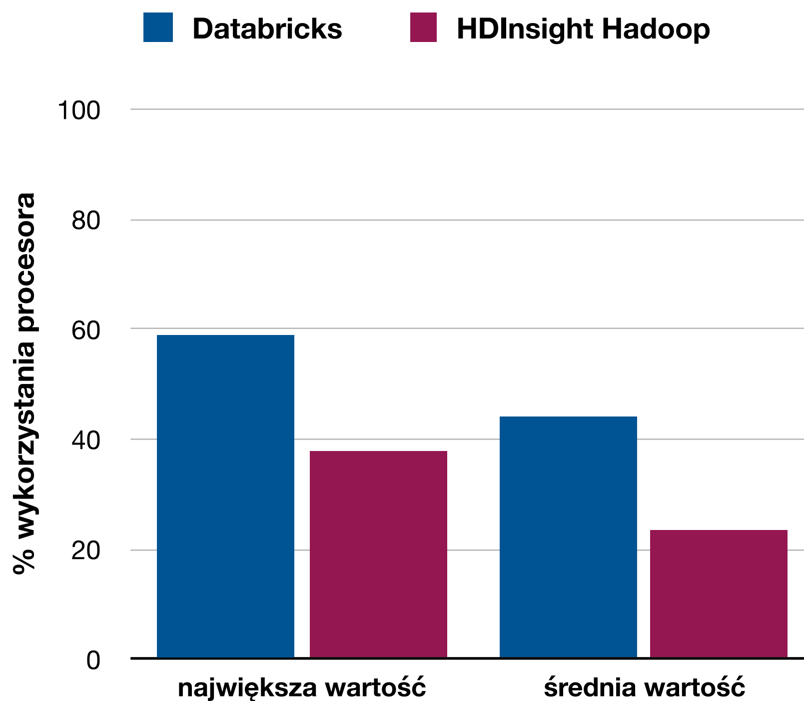
Wyznaczanie π	**Databricks**	**HDInsight**
Całkowity czas – mediana	5 min 17 s	9 min 47 s
Całkowity czas – średnia	4 min 57 s	9 min 39 s
Odchylenie standardowe od średniej	47,33 s	16,22 s

Tabela 7.4. Wyznaczanie liczby π – wyniki dla 5 uruchomień

Wyznaczanie π	Databricks	HDInsight
Całkowity czas – mediana	4 min 6 s	14 min 4 s
Całkowity czas – średnia	4 min 27 s	14 min 57 s
Odchylenie standardowe od średniej	45,18 s	122,6 s

Tabela 7.5. Przetwarzanie danych sklepu internetowego – wyniki dla 5 uruchomień

Kolejną analizowaną metryką jest procentowe wykorzystanie procesora. Na wykresie 7.3. przedstawiono porównanie średniego i maksymalnego wykorzystania procesora. Wartości dla obu klastrów nie są wysokie, trzeba jednak zaznaczyć, że metryka ta wyliczana jest dla wszystkich węzłów w klastrze – zarówno tych wykonujących operacje (ang. *worker nodes*), jak i węzła głównego (ang. *master node*), który zajmuje się rozdzielaniem zasobów i zadań i nie wykonuje obliczeń. Mając jedynie dwa programy uruchomione jednocześnie i brak potrzeby dynamicznego rozdzielania zasobów, wykorzystanie procesora w węźle głównym nie jest wysokie, co obniża ogólne statystyki.



Rys. 7.3. Mediana dla 5 uruchomień, procentowe wykorzystanie procesora

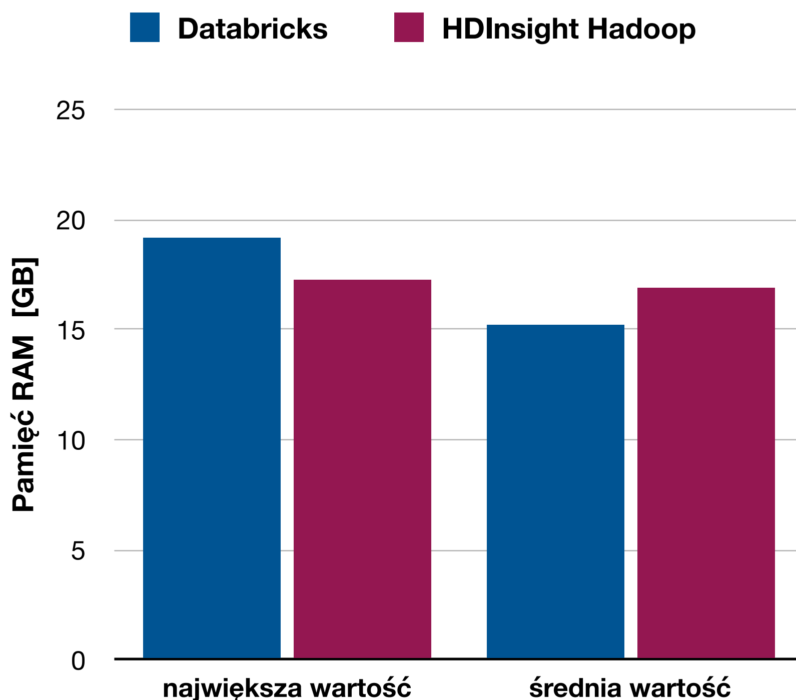
Dla klastra HDInsight procentowe wykorzystanie procesora jest wyraźnie niższe niż dla Databricks. Przyczyną takiej sytuacji może być złe rozdzielanie równoległych zadań między węzły – jeśli któryś z węzłów musi czekać na wyniki operacji, to pozostaje bezczynny. Problemu szukać można również w konfiguracji kalkulatora zasobów [49] w YARN, który domyślnie alokuje zasoby wyłącznie w oparciu o pamięć RAM. Możliwe jest użycie w konfiguracji opcji

`DominantResourceCalculator`, która bierze pod uwagę zarówno pamięć, jak i wykorzystanie procesora. Taka konfiguracja pozwala na przydzielanie większej ilości zasobów procesora zadaniom, które są bardziej wymagające obliczeniowo. Opcja ta nie została sprawdzona w przykładzie praktycznym, więc powyższe rozważania są teoretyczne. Podczas tworzenia nowego klastra HDInsight możliwe jest również włączenie opcji automatycznego skalowania klastra [50], która pozwala na zwiększanie lub zmniejszanie liczby węzłów na podstawie wybranych wcześniej kryteriów. Skalowanie może być wywoływane po przekroczeniu jakiegoś progu obciążenia klastra (ang. *load-based scaling*) lub o określonych przez użytkownika porach (ang. *schedule-based scaling*), w których regularnie wykonywane są operacje. Pozwala to na oszczędzanie zasobów i używanie ich jedynie w razie potrzeby, co obniża koszty. Nie jest możliwe jednak skalowanie liczby rdzeni w klastrze i pamięci RAM, co sprawia, że użytkownik musi w tej kwestii podjąć odpowiednią decyzję na etapie tworzenia klastra.

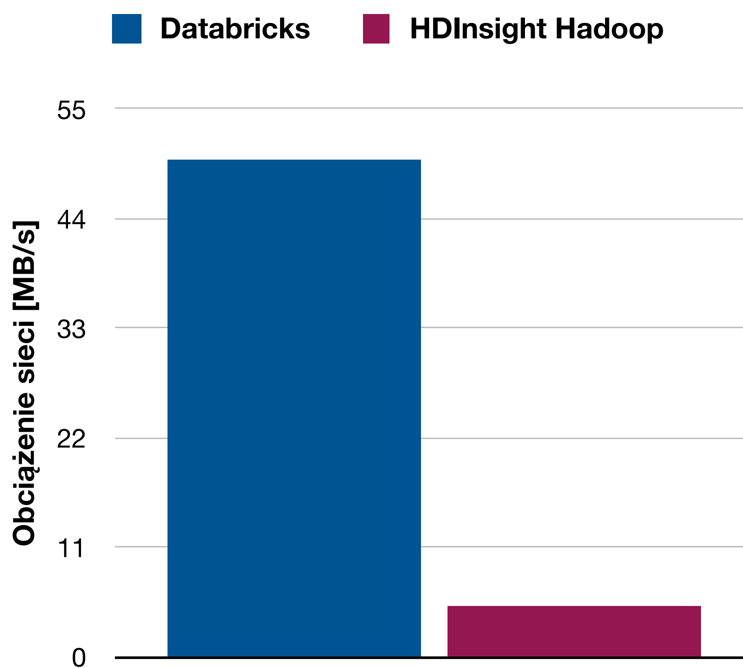
W Databricks wykorzystanie procesora jest większe, choć pewna część pozostaje bezczynna przez większość czasu. W tym przypadku opcją optymalizacji również jest automatyczne skalowanie klastra, choć nie ma możliwości wybierania sposobu wywoływania skalowania – jest ono zawsze dopasowywane do aktualnego procentowego obciążenia.

Oprócz obciążenia procesora, warto przeanalizować również średnie obciążenie dla jednej minuty, będące średnią liczbą procesów w kolejce, które oczekują na wykonanie (w tym zawarte są również procesy aktualnie wykonywane). Jeśli obciążenie procesora nie jest maksymalne, a mimo to wartość średniego obciążenia minutowego przekracza łączną liczbę rdzeni we wszystkich węzłach, to wskazuje to na problemy z wydajnością systemu dysków. Oba analizowane klastry mają po 16 rdzeni mogących wykonywać operacje, natomiast mediana średniego obciążenia dla jednej minuty wyniosła 4,8 dla Databricks i 2,92 dla HDInsight. W obu przypadkach nie ma więc problemów z wydajnością, a większość rdzeni pozostaje bezczynna. Widzimy więc, że możliwości optymalizacyjnych należy szukać w sposobie rozdziału i liczbie równoległych zadań.

Bardzo istotnym aspektem jest również wykorzystanie pamięci RAM. Wykres 7.4. przedstawia porównanie największego i średniego użycia dla obu klastrów. Widzimy, że o ile średnia wartość jest nieco niższa dla klastra Databricks, to osiąga on większe maksymalne zużycie RAM niż klaster HDInsight. Jeśli wyliczymy średnie całkowite zużycie pamięci w czasie, to wynik klastra Databricks wynosi 4 818,4 GB $\cdot$ s, natomiast dla HDInsight – 14 263, 6 GB $\cdot$ s. Różnica jest wyraźna i można tę wartość uznać za bardziej miarodajną niż samo średnie wykorzystanie pamięci, bo pokazuje również, jak długo klaster jest obciążony. Zasadniczo im mniejsze wykorzystanie pamięci RAM, tym lepiej, bo można użyć mniej kosztownych klastrów. Często jednak trudno jest z góry określić, ile pamięci RAM będzie potrzebne. Kolejny raz więc zaletą jest opcja automatycznego skalowania klastra, która pozwala na oszczędzenie pamięci przez zmniejszenie liczby węzłów.



Rys. 7.4. Mediana dla 5 uruchomień, wykorzystanie pamięci RAM



Rys. 7.5. Mediana dla 5 uruchomień, obciążenie sieci dla danych przychodzących

Ostatnią analizowaną metryką jest obciążenie sieci mierzone w MB/s. Wykres 7.5. przedstawia porównanie średniego obciążenia dla danych przychodzących. Widzimy tu wyraźną równicę na korzyść HDInsight, gdyż Hadoop korzysta z plików zapisanych w systemie plików HDFS, który

jest zlokalizowany w magazynie Azure Blob Storage. Nie ma więc konieczności dodatkowego łączenia się z tym magazynem ani rozsyłania danych między węzły, co przekłada się na niższe obciążenie sieci. Databricks natomiast, jeśli nie korzystamy z plików i tabel w DBFS, musi się łączyć z magazynem Blob Storage, co powoduje większe obciążenie sieci. Przeanalizowane zostało również średnie obciążenie sieci dla danych wyjściowych, ale z racji małego rozmiaru plików wyjściowych, te wartości nie były wysokie i wyniosły: 157,1 KB/s dla Databricks i 198 KB/s dla HDInsight.

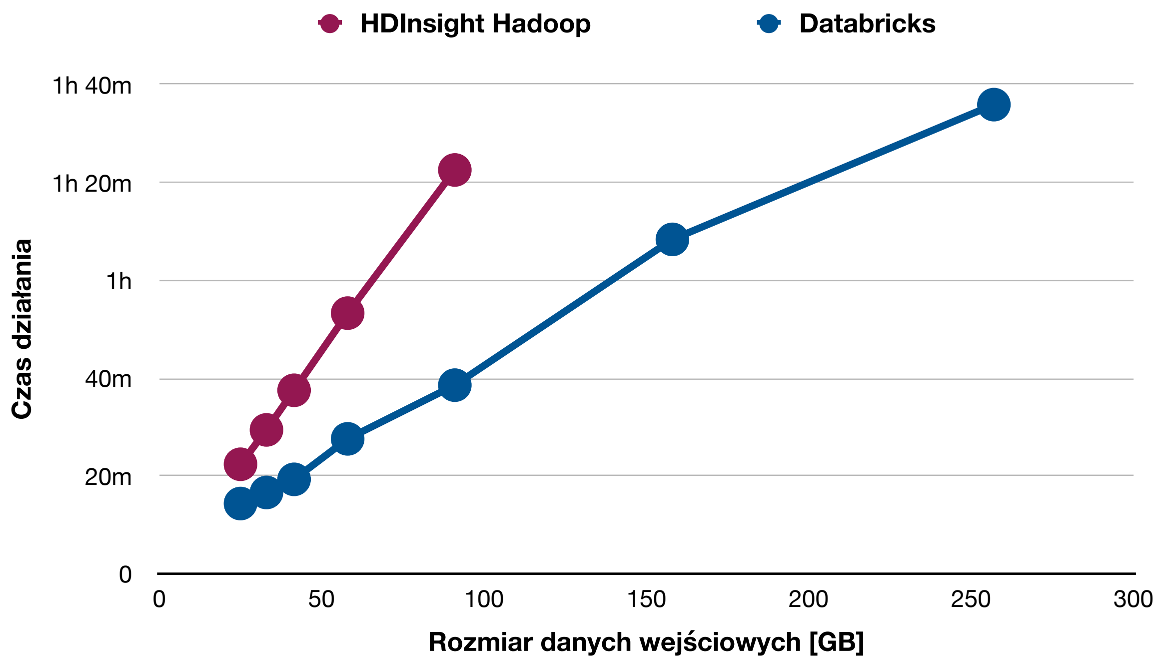
7.3 Skalowalność

W tej części przeanalizowany został wpływ rozmiaru danych na czas działania zadania na poszczególnych klastrach. Jak dotąd we wszystkich porównaniach czasowych lepiej wypadał klaster Databricks. Jednak podczas testowania architektury lokalnej, podczas zwiększania rozmiaru danych, lepiej od klastra Sparka wypadł klaster Hadoopa. Sprawdzono więc, czy taka sama zależność występuje po przeniesieniu tych technologii do chmury. Do porównania użyto programu do zliczania słów w plikach tekstowych. Zbiór danych był stopniowo zwiększany przy każdym kolejnym uruchomieniu programu i składał się z plików tekstowych o zróżnicowanych rozmiarach: 2,4 MB, 91 MB, 426 MB, 560 MB oraz 7,3 GB. Z powodu wysokich kosztów tych operacji, dla każdej opcji wykonana została tylko jedna próba. Otrzymane wyniki mogą się różnić o kilka minut od typowych, ale różnice pomiędzy czasami działania programu dla poszczególnych klastrów są na tyle duże, że wyznaczenie mediany z większej liczby uruchomień nie zmieniłoby znacząco dynamiki wykresu.

Wykres 7.6. przedstawia czas działania programu dla siedmiu różnych rozmiarów zbioru danych wejściowych. Dla klastra HDInsight nie zostały przeprowadzone pomiary dla danych o wielkości 158 GB oraz 257 GB. Widzimy, że o ile czasy działania programu dla mniejszych rozmiarów danych są stosunkowo zbliżone do siebie, to wraz ze zwiększaniem zbioru danych, wykresy się rozbiegają coraz bardziej. Dla danych wejściowych o rozmiarze 91 GB, czas działania programu na klastrze HDInsight wynosi o 44 min więcej, niż na klastrze Databricks.

Różnice wystąpiły również w trudności uruchomienia przykładów. Na klastrze Databricks program do zliczania słów zadziałał bez konieczności wprowadzania zmian w kodzie programu uruchamianego wcześniej lokalnie, na klastrze Sparka. Natomiast do programu korzystającego z Hadoop MapReduce konieczne było wprowadzenie pewnych zmian, gdyż program się zawieszał w pewnym momencie. Powodem takiej sytuacji była zbyt duża ilość danych przekazywana przez funkcje mapujące do funkcji redukującej. Aby tego uniknąć, należało skorzystać z narzędzia optymalizującego zwanego łącznikiem [51] (ang. *combiner*). Jest to dodatkowa klasa, która agreguje dane dla tych samych kluczy z każdego zadania mapującego, dzięki czemu do funkcji redukującej dociera mniej danych i dokonuje ona sumowania wartości jedynie pomiędzy różnymi zadaniami mapującymi. Kolejnym napotkanym problemem było przetwarzanie małych plików

– w wejściowych zbiorach danych znajdowało się kilkanaście lub kilkadziesiąt plików o rozmiarach 2,4 MB. Ich przetwarzanie znacznie wydłuża czas działania zadania zaimplementowanego w Hadoop MapReduce. W rozdziale 7.1. przedstawione zostało rozwiązanie tego problemu, czyli użycie `CombineFileInputFormat`. W tym przypadku jednak opcja ta się nie sprawdza – rozwiązanie to co prawda łączy skutecznie małe pliki, ale utyka na bardzo dużych. Najbardziej skutecznym rozwiązaniem tego problemu jest zapisanie plików do bazy danych, dzięki czemu kod przetwarzający dane nie musi uwzględniać rozmiaru plików.



Rys. 7.6. Czas działania programu do zliczania słów w zależności od rozmiaru danych wejściowych

Widzimy więc, że korzystanie z Databricks jest znacznie prostsze i wymaga mniejszej wiedzy, gdyż klaster dokonuje podstawowej optymalizacji za użytkownika. W powyższych próbach klaster był za każdym razem restartowany przed uruchomieniem programu, więc nie skorzystano z zalet pamięci podręcznej w Databricks. Jednak gdyby ten sam program uruchamiany był kilkukrotnie, bez restartowania klastra, to czas drugiego i kolejnego uruchomienia byłby krótszy.

Klaster HDInsight posiada bardzo wiele opcji i parametrów do konfigurowania. Domyślne wartości nie zawsze są optymalne, warto więc samodzielnie dostosować poszczególne parametry. Wymaga to jednak wiedzy o wpływie i zastosowaniu poszczególnych parametrów, więc zwykle konieczna jest najpierw identyfikacja problemu, a potem zmiana parametrów. Przykładowo, w analizowanym tu programie wystąpił problem z przepełnieniem stosu Javy, konieczne było więc przydzielenie mu w konfiguracji większej ilości pamięci RAM. Niewykluczone więc, że przy odpowiedniej konfiguracji i po połączeniu małych plików, dałoby się na klastrze HDInsight osiągnąć podobne czasy działania programu, jakie uzyskano dla klastra Databricks. Wymaga to jednak wiedzy, licznych prób i sporego nakładu pracy, więc niekoniecznie jest to opłacalne.

7.4 Podsumowanie

W niniejszym rozdziale porównane były jedynie poszczególne rozwiązania chmurowe, więc otrzymane wyniki i wnioski nie obejmują wszystkich możliwych zastosowań i platform, w których używane są technologie Spark i Hadoop. Niewątpliwie platforma Databricks jest godnym uwagi narzędziem – została stworzona i zoptymalizowana przez twórców Apache Spark, więc rozwija wszystkie jego zalety i wzbogaca o nowe możliwości. Plusem jest również łatwość użycia i automatyczna optymalizacja klastra bez konieczności konfigurowania go dodatkowo przez użytkownika. Dodatkowo w systemie plików DBFS można tworzyć własne tabele i przenosić dane, które później są znaczenie szybciej odczytywane przez klaster, niż te zapisane w magazynie Azure.

Usługa HDInsight posiada kilka różnych typów klastrów – w tej pracy przeanalizowany został jedynie klaster korzystający z Apache Hadoop. Widzimy tu pewną przestarzałość tej technologii – dużo opcji konfiguracyjnych pozwala co prawda na optymalizację działania programu, jednak płacąc za usługę, można oczekiwać, że uruchomiony program będzie działał możliwie najlepiej z domyślnymi ustawieniami. Dodatkowo HDInsight nie uwydatnia podstawowej zalety technologii Hadoop, którą jest wydajne przetwarzanie naraz dużych ilości danych. Plusem klastra HDInsight jest z pewnością wbudowany interfejs Hive, który zapewnia wygodny widok do wprowadzania zapytań SQL i tworzenia tabel, a także szybkiego przenoszenia danych z HDFS do folderu Hive. Opcja ta jest jednak dostępna jedynie dla HDInsight w wersji 3.6. Dostępna jest już nowa wersja – 4.0, która nie posiada takiej opcji i korzystanie z bazy danych i widoku Hive możliwe jest jedynie za pomocą konsoli, co nie jest ani przejrzyste, ani wygodne.

Opłaty za oba klastry są naliczane za ich czas użycia, więc czas działania programów i dobra utylizacja zasobów grają tu główną rolę. Jeśli programy działają długo, a nie wykorzystują mocy obliczeniowej i pamięci we wszystkich węzłach w klastrze, to generuje to niepotrzebne koszty. Warto więc podczas tworzenia klastra włączyć opcję automatycznego skalowania klastra, która jest dostępna zarówno w Databricks, jak i w HDInsight.

8. Wnioski

Badania przeprowadzone w niniejszej pracy dostarczyły wielu ciekawych obserwacji – niektóre z nich były spodziewane, ale występowały też wyniki zaskakujące. Szczegółowa analiza metryk, a także sposobu działania poszczególnych aplikacji pozwoliła na znalezienie wyjaśnień różnych sytuacji, które celowo bądź przypadkowo wynikły podczas przeprowadzania testów. Doprowadziło to do uzyskania nowej wiedzy, która może być wykorzystana podczas tworzenia systemu big data.

W pracy porównane zostały dwie technologie do przetwarzania danych – Apache Spark oraz Apache Hadoop oraz ich użycie w chmurze Azure. Wykonana analiza pokazała, jak wiele możliwości i zalet oferuje Spark oraz że jest to skuteczne narzędzie do przetwarzania danych, które zyskuje dodatkowe możliwości optymalizacyjne w połączeniu z usługą Databricks. Gorzej w porównaniu wypadł Hadoop – o ile sam klaster ma wiele zalet i uruchamianie na nim programów korzystających z Apache Spark może przynosić korzyści związane ze skalowalnością i łatwym łączeniem z innymi usługami z ekosystemu Apache Hadoop, to już sam framework MapReduce wydaje się zbyt przestarzały dla współczesnego programisty. Posiada wiele ograniczeń związanych z wykorzystywanymi funkcjami, konieczne jest także zapisanie problemu w formie mapowania i redukowania, co zwykle znacznie wydłuża kod. Do tego Hadoop bardzo źle działa z małymi plikami, więc tę kwestię trzeba zawsze uwzględniać, chyba że mamy pewność, że przetwarzane są wyłącznie duże pliki.

Po sprawdzeniu działania poszczególnych technologii w chmurze Azure, Hadoop wypada jeszcze gorzej – HDInsight nie podkreśla jego zalet, tak jak to się dzieje w przypadku Sparka i Databricks. Powodem tego może być ogólność usługi HDInsight, która obsługuje różne klastry i nie została stworzona specjalnie dla Hadoopa. Wiele innych dostawców chmurowych oferuje klastry i środowisko dla Apache Hadoop, więc pracę tę można rozwinąć o porównanie innych opcji uruchomienia w chmurze. Technologia ta z pewnością nadal będzie używana – wiele firm oparło działanie swoich systemów na architekturze lambda, w której Hadoop zajmuje znaczące miejsce. Korzystanie z tej technologii wymaga jednak znacznej wiedzy – przedsiębiorstwa dopiero zaczynające z big data mogą chętniej wybrać architekturę kappa i skorzystać z systemu do przetwarzania danych strumieniowych opartego na frameworku Spark. Wnioski wyciągnięte z niniejszej pracy zachęcają do takich wyborów – narzędzie Databricks jest łatwe w użyciu i pozwala na zoptymalizowane uruchamianie zadań.

Podsumowując, cel pracy został osiągnięty. Udało się zrealizować przykłady, które pozwoliły na pokazanie wad i zalet poszczególnych rozwiązań. Pokazano również możliwości optymalizacyjne, które nie tylko poprawiają czas działania programu i wykorzystanie zasobów, ale także pozwalają uniknąć wystąpienia błędów podczas przetwarzania danych.

Bibliografia

[1] Lu Tan i Neng Wang. „Future internet: The Internet of Things”. W: *2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE)* 5 (2010), s. 376–380.

[2] *Where does ‘Big Data’ come from?* *https://www.bigdataframework.org/short-history-of-big-data*. [online; stan na 21.07.2020]. 2019.

[3] *The 4 Characteristics of Big Data.* *https://www.bigdataframework.org/four-vs-of-big-data*. [online; stan na 21.07.2020]. 2019.

[4] Chandrima Sinha Roy, Siddharth S. Rautaray i Manjusha Pandey. „Big Data Optimization Techniques: A Survey”. W: *International Journal of Information Engineering and Electronic Business* 10 (2018), s. 41–48.

[5] Christine Taylor. *Big Data Architecture.* *https://www.datamation.com/big-data/big-data-architecture.html*. [online; stan na 14.07.2020]. 2017.

[6] *Big Data Architectures.* *https://docs.microsoft.com/pl-pl/azure/architecture/data-guide/big-data*. [online; stan na 14.07.2020]. 2018.

[7] *What Is Lambda Architecture?* *https://hazelcast.com/glossary/lambda-architecture*. [online; stan na 14.07.2020].

[8] *What Is the Kappa Architecture?* *https://hazelcast.com/glossary/kappa-architecture*. [online; stan na 14.07.2020].

[9] Jeffrey Dean i Sanjay Ghemawat. „MapReduce: Simplified Data Processing on Large Clusters”. W: *OSDI’04: Sixth Symposium on Operating System Design and Implementation* (2004), s. 137–150.

[10] Adam Kajstura. *Metoda k-średnich.* *https://www.statystyka.az.pl/analiza-skupien/metoda-k-srednich.php*. [online; stan na 14.07.2020].

[11] *Apache Hadoop. https://hadoop.apache.org.*
[online; stan na 14.07.2020].

[12] *HDFS Architecture. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.*
[online; stan na 14.07.2020]. 2019.

[13] *Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.*
[online; stan na 14.07.2020]. 2019.

[14] Grzegorz Malewicz i in. „Pregel: A System for Large-Scale Graph Processing". W: *Proceedings of the 2010 international conference on Management of data* (2010), s. 135–146.

[15] Yingyi Bu i in. „HaLoop: Efficient Iterative Data Processing on Large Clusters". W: *Proc. VLDB Endow.* 3 (2010), s. 285–296.

[16] Matei Zaharia i in. „Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". W: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (2012), s. 15–28.

[17] *Apache Spark. http://spark.apache.org.*
[online; stan na 14.07.2020].

[18] Jules Damji. *A Tale of Three Apache Spark APIs: RDDs vs DataFrames and Datasets. https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html.*
[online; stan na 14.07.2020]. 2016.

[19] Mark Litwintschik. *Is Hadoop Dead? https://tech.marksblogg.com/is-hadoop-dead.html.*
[online; stan na 14.07.2020]. 2019.

[20] *Google Trends. https://trends.google.com/trends.*
[online; stan na 14.07.2020].

[21] Mohamed Ali Ismail i in. „Measuring the Performance of Big Data Analytics Process". W: *Journal of Theoretical and Applied Information Technology* (2019), s. 3796–3808.

[22] *Garbage Collection Tuning Guide. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning.*
[online; stan na 14.07.2020].

[23] *Apache HBase. https://hbase.apache.org.*
[online; stan na 14.07.2020].

[24] *Apache Hive. https://hive.apache.org.*
[online; stan na 14.07.2020].

[25] *Hadoop Ecosystem. https://databricks.com/glossary/hadoop-ecosystem.* [online; stan na 14.07.2020].

[26] *Cluster Mode Overview. https://spark.apache.org/docs/latest/cluster-overview.html.* [online; stan na 14.07.2020].

[27] *Apache Cassandra. https://cassandra.apache.org.* [online; stan na 14.07.2020].

[28] *The World's Top Cloud Vendors. https://cloudwars.co/cloud-wars-top-10-vendors-world.* [online; stan na 14.07.2020]. 2020.

[29] *Microsoft Azure. https://azure.microsoft.com.* [online; stan na 14.07.2020].

[30] *HDInsight. https://azure.microsoft.com/pl-pl/services/hdinsight.* [online; stan na 14.07.2020].

[31] *Databricks. https://databricks.com.* [online; stan na 14.07.2020].

[32] *Azure Blob Storage. https://azure.microsoft.com/pl-pl/services/storage/blobs.* [online; stan na 14.07.2020].

[33] Ali Ghodsi i Rohan Kumar. *Azure Databricks, industry-leading analytics platform powered by Apache Spark. https://databricks.com/blog/2018/03/22/azure-databricks-industry-leading-analytics-platform-powered-by-apache-spark.html.* [online; stan na 14.07.2020]. 2018.

[34] Martin Odersky, Lex Spoon i Bill Venners. *Programming in Scala.* Artima, 2008.

[35] Alicja Luszczak i in. *Databricks Cache Boosts Apache Spark Performance. https://databricks.com/blog/2018/01/09/databricks-cache-boosts-apache-spark-performance.html.* [online; stan na 14.07.2020]. 2018.

[36] *Wolne Lektury. https://wolnelektury.pl.* [online; stan na 14.07.2020].

[37] *Data Blocks. http://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.* [online; stan na 23.07.2020].

[38] *Disk space versus namespace. https://docs.cloudera.com/runtime/7.1.0/hdfs-overview/topics/hdfs-disk-space-versus-namespace.html.* [online; stan na 14.07.2020].

[39] *Running Spark on YARN. https://spark.apache.org/docs/latest/running-on-yarn.html.* [online; stan na 14.07.2020].

[40] *have i been pwned?* *https://haveibeenpwned.com*.
[online; stan na 14.07.2020].

[41] *Level of parallelism.* *http://spark.apache.org/docs/latest/tuning.html#level-of-parallelism*.
[online; stan na 14.07.2020].

[42] *Apache Parquet.* *https://parquet.apache.org/documentation/latest*.
[online; stan na 14.07.2020].

[43] *What is Hadoop Streaming? Explore How Streaming Works.* *https://data-flair.training/blogs/hadoop-streaming*.
[online; stan na 14.07.2020]. 2020.

[44] *Overview of Apache Spark Streaming.* *https://docs.microsoft.com/pl-pl/azure/hdinsight/spark/apache-spark-streaming-overview*.
[online; stan na 14.07.2020]. 2020.

[45] *Apache Storm.* *https://storm.apache.org*.
[online; stan na 25.07.2020].

[46] *Apache Flink.* *https://flink.apache.org*.
[online; stan na 25.07.2020].

[47] Michael Kechinov. *eCommerce behavior data from multi category store.* *https://www.kaggle.com/mkechinov/ecommerce-behavior-data-from-multi-category-store*.
[online; stan na 14.07.2020]. 2019.

[48] *Databricks File System (DBFS).* *https://docs.databricks.com/data/databricks-file-system.html*.
[online; stan na 14.07.2020]. 2020.

[49] *CPU Scheduling.* *https://docs.cloudera.com/HDPDocuments/HDP2/HDP-2.4.0/bk_yarn_resource_mgt/content/ch_cpu_scheduling.html*.
[online; stan na 25.07.2020].

[50] *Automatically scale Azure HDInsight clusters.* *https://docs.microsoft.com/en-us/azure/hdinsight/hdinsight-autoscale-clusters*.
[online; stan na 25.07.2020]. 2020.

[51] *Combiner in Hadoop MapReduce.* *https://knpcode.com/hadoop/mapreduce/combiner-in-hadoop-mapreduce*.
[online; stan na 25.07.2020]. 2018.