Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Mateusz Kiebała
Nr albumu: 359758

Biblioteka do implementacji algorytmów minimalnych

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr Jacka Sroki

Październik 2019

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

Streszczenie

W ramach pracy magisterskiej została stworzona biblioteka umożliwiająca implementowanie rozproszonych algorytmów minimalnych. Wspiera ona dwa główne frameworki służące do pisania programów rozproszonych: Hadoop i Spark. Celem biblioteki jest udostępnienie API umożliwiającego optymalne i łatwe implementowanie algorytmów minimalnych. W ramach pracy powstały również implementacje przykładowych algorytmów minimalnych takich jak: tworzenie rankingu, statystyki prefiksowe, grupowanie, pół-złączenia (ang. *semi-join*) oraz statystyka okienkowa (ang. *sliding aggregation*).

Słowa kluczowe

Spark, Hadoop, MapReduce, algorytmy minimalne, algorytmy rozproszone, programowanie rozproszone, TeraSort, big data

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

Software and its engineering – Software notations and tools
Software and its engineering – Cloud computing
Software and its engineering – Frameworks
Theory of computation — Design and analysis of algorithms
Theory of computation – MapReduce algorithms
Computer systems organization – Cloud computing

Spis treści

Rozdział 1

Wprowadzenie

W obecnych czasach programowanie rozproszone jest prężnie rozwijającą się dziedziną informatyki. Rozwój technologii i nauki doprowadził do gwałtownego wzrostu gromadzonych danych w różnych dziedzinach życia. Rodzi to potrzebę efektywnego przetwarzania ogromnych zbiorów informacji [11, 13].

Obecnie dwoma najbardziej popularnymi frameworkami do przetwarzania dużych danych są Hadoop i Spark [7]. Pierwszy z nich opiera się na paradygmacie MapReduce [12], natomiast drugi korzysta z rozproszonych kolekcji danych ze zbiorem typowych operatorów do ich przetwarzania [14]. W obu systemach algorytm rozproszony uruchamiany jest na grupie niezależnych maszyn, komunikujących się za pomocą sieci, zwanych klastrem obliczeniowym.

Algorytm oparty o paradygmat MapReduce wykonuje się w rundach, a każda runda składa się z trzech faz: *map*, *shuffle* i *reduce*. Podczas faz *map* i *reduce* nie występuje komunikacja między maszynami. W trakcie fazy *map* dane zostają wczytane i wstępnie przetworzone. Następnie faza *shuffle* dba o pogrupowanie i prawidłowe rozesłanie wyników fazy *map* na pamięć lokalną poszczególnych maszyn. W fazie końcowej *reduce*, obliczany jest wynik na podstawie pogrupowanych danych z poprzedniej fazy. Algorytm może posiadać wiele rund. Danymi wejściowymi rundy *i+1* jest wynik działania rundy *i*. Dane pomiędzy rundami są wymieniane poprzez HDFS, co jest kosztownym rozwiązaniem. [8, 10, 12].

Spark opiera swoje działanie na elastycznych, rozproszonych zestawach danych (ang. *Resilient Distributed Dataset - RDD*). RDD są niezmienialnymi kolekcjami danych trzymanymi w pamięci podręcznej lub na dyskach lokalnych maszyn. Dzięki bogatemu API operowanie na RDD jest bardzo intuicyjne i efektywne. Nowe RDD tworzone są przez transformacje istniejących [14].

Projektując algorytmy działające na Hadoopie i Sparku, trzeba zwrócić uwagę na: równoważenie obciążenia maszyn, zużycie pamięci i CPU, ilość wykonywanych operacji odczytu i zapisu do plików oraz przesył danych między maszynami. Z tego powodu powstała klasa algorytmów minimalnych, czyli wzorzec algorytmu do którego powinniśmy dążyć. Są to algorytmy rozproszone, gwarantujące zrównoważone zużycie pamięci na każdej z maszyn, ograniczony przesył danych między maszynami, zakończenie algorytmu po stałej liczbie rund / transformacji oraz przyspieszenie obliczeń skalujące się liniowo względem liczby maszyn [9].

1.1. Motywacja

Motywacją do napisania biblioteki jest fakt, że spora grupa algorytmów minimalnych opisanych w literaturze bazuje na tych samych obliczeniach początkowych, np. na posortowaniu i zrównoważonym podzieleniu danych, a następnie na przesyłaniu między serwerami ograni-

czonych statystyk. Celem biblioteki jest udostępnienie optymalnej i łatwej w użyciu bazy algorytmów minimalnych, tak aby zwolnić użytkownika od powtarzalnych czynności przy ich implementowaniu i pozwolić mu skupić się na unikatowej części algorytmu.

Biblioteka do implementacji algorytmów minimalnych redukuje ilość pisanego kodu. W przypadku Sparka jest niewielka, ze względu na zwięzłość samego frameworka i bogatego API, jednak w Hadoopie jest bardzo widoczna. Głównie przyczyniają się do tego generalizacja faz MapReduce oraz format danych Avro. Sama tylko możliwość posortowania dowolnych obiektów ma ogromne znaczenie w redukcji kodu. Dodatkowo wszystkie fazy MapReduce i funkcje udostępniane przez API są generyczne. Wystarczy zdefiniować klasę obiektów, a reszta implementacji pozostaje stała. Zarówno w Sparku jak i w Hadoopie, do porównywania obiektów używane są komparatory, dzięki czemu użytkownik jeszcze mniejszym nakładem czasu i pracy może testować różne wersje algorytmu.

Ważnym aspektem rozwiązanym przez bibliotekę jest zarządzanie maszynami. Dzięki abstrakcyjnej numeracji maszyn biblioteka umożliwia przesyłanie danych na dowolnie wybrane z nich. Pozwala to na bardziej precyzyjne zarządzanie transferem danych.

Następną zaletą biblioteki jest ułatwienie przesyłania wielu typów obiektów między fazami i rundami MapReduce w Hadoopie oraz między transformacjami RDD na Sparku. Niektóre algorytmy rozproszone, np. *lista rankingowa* 3.3, *statystyki prefiksowe* 3.4 wymagają przesyłania wielu rodzajów obiektów jednocześnie. Niestety podstawowe wersje Hadoopa i Sparka nie wspierają takiego rozwiązania.

Biblioteka pozwala także na definiowanie własnych funkcji statystyk łącznych na zbiorze obiektów. Razem z zaimplementowanymi metodami operującymi na statystykach, biblioteka pozwala na efektywne obliczanie skomplikowanych statystyk oraz testowanie wielu z nich przy jednorazowej implementacji algorytmu.

Rozdział 2

Podstawowe pojęcia

2.1. Hadoop

Podrozdział został napisany w oparciu o [8, 10, 12]. Hadoop jest otwartym frameworkiem, opartym o język Java, pozwalającym na rozproszone przechowywanie i przetwarzanie dużych zbiorów danych. W tym celu wykorzystuje prosty model programistyczny i rozprasza obliczenia na klastry. Został zaprojektowany z myślą łatwego skalowania. Działa doskonale zarówno na jednej jak i tysiącach maszynach. Dynamiczne dodawanie nowych komputerów do klastra jest łatwe i nieinwazyjne dla działającego systemu. Platforma Hadoop wykorzystuje technikę replikacji danych między maszynami, dzięki czemu zapewniony jest spójny i ciągły dostęp do danych, nawet w momencie awarii któregoś z serwerów. Dodatkowo Hadoop wykrywa i zarządza błędami warstwy aplikacji. Dzięki temu użytkownik nie musi polegać już na niezawodności sprzętu komputerowego. W skład podstawowej wersji Hadoop wchodzą:

- Hadoop Distributed File System (HDFS) – rozproszony systemem plików (patrz punkt 2.1.1),
- Hadoop Common – zbiór bibliotek (patrz punkt 2.1.2),
- Hadoop MapReduce – framework do implementacji aplikacji rozproszonych (patrz punkt 2.1.3),
- YARN – system zarządzania zasobami klastra (patrz punkt 2.1.4).

Oprócz wymienionych wyżej modułów istnieją także w pełni zintegrowane rozszerzenia, ułatwiające zarządzanie danymi oraz usługami klastra. Poza otwartym rozwiązaniem, które skupiło wokół siebie sporą grupę kontrybutorów, istnieją także dystrybucje komercyjne. Posiadają one dodatkowe narzędzia tworzące gotowy serwis do przetwarzania dużych danych. Dodatkową zaletą takich rozwiązań jest wsparcie całego ekosystemu Hadoop, a nie tylko poszczególnych modułow.

2.1.1. Hadoop Distributed File System (HDFS)

HDFS to rozproszony systemem plików, przeznaczony do przechowywania ogromnych zbiorów danych. Wyróżniają go odporność na awarie serwerów oraz wysoka jakość pracy na niskobudżetowym sprzęcie komputerowym. Obecnie klastry składają się z tysięcy maszyn. Każda z nich ma niezerowe prawdopodobieństwo zepsucia się. W praktyce oznacza to, że często któraś z maszyn nie działa. Dlatego też, wykrywanie, reagowanie i szybkie naprawianie usterek jest fundamentalną częścią HDFS.

Architektura HDFS składa się z serwera nadzorującego (*NameNode*) oraz maszyn wykonawczych (*DataNodes*). Dane są zapisywane w postaci plików. HDFS jest systemem niezawodnym w kontekście przechowywania danych. Każdy plik jest trzymany jako sekwencja bloków, gdzie wszystkie oprócz ostatniego są tej samej wielkości. Bloki natomiast są replikowane i przechowywane na wielu *DataNodes*. *NameNode* jest odpowiedzialny za otwieranie, zamykanie i nazywanie plików i folderów. Zarządza także mapowaniem bloków plików na *DataNodes*. *DataNodes* są natomiast odpowiedzialne za tworzenie, usuwanie i replikację bloków oraz za obsługiwanie poleceń czytania i pisania do bloków.

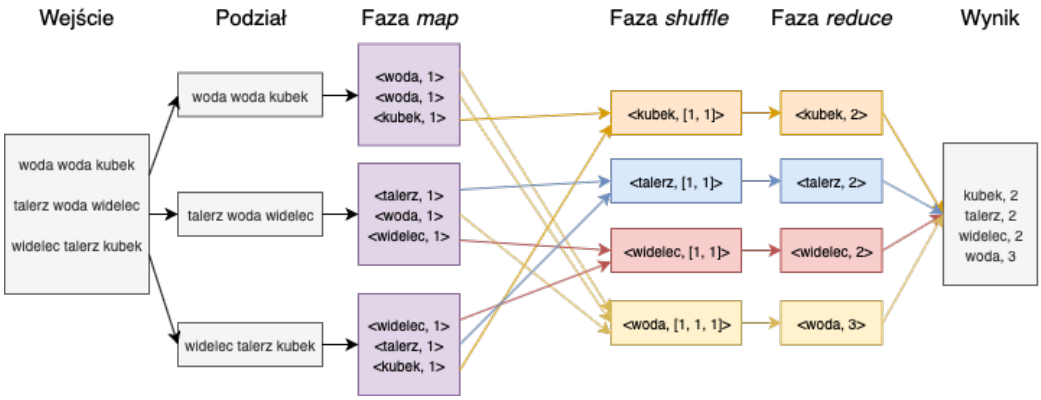
2.1.2. Hadoop Common

Hadoop Common jest zbiorem bibliotek wspierającym i integrującym pozostałe moduły Hadoopa. Tworzy warstwę abstrakcji do zarządzania systemem oraz dostępem do HDFS. Zawiera także niezbędne pliki JAR oraz skrypty pozwalające na uruchomienie Hadoopa. Dodatkowo udostępnia kod źródłowy oraz dokumentację Hadoopa.

2.1.3. Hadoop MapReduce

Hadoop MapReduce jest frameworkiem opartym o paradygmat MapReduce, służącym do łatwego implementowania aplikacji rozproszonych, operujących na ogromnych zbiorach danych. Pozwala na przetwarzanie nieustrukturyzowanych danych, zapewniającym przy tym niezawodność oraz odporność programu na awarie.

MapReduce operuje wyłącznie na parach *<klucz, wartość>*. Typy *klucz* oraz *wartość* muszą być serializowalne przez framework oraz muszą implementować komparatory. Początkowo główny proces MapReduce dzieli dane wejściowe na niezależne paczki, które są następnie przetwarzane w sposób równoległy przez procesy *map*. Kolejnym krokiem jest faza *shuffle*, w której następuje pogrupowanie i rozesłanie wyników fazy *map*. Warto zauważyć, że obiekty o takich samych kluczach trafiają na te same maszyny. Tak przygotowane dane stają się wejściem dla procesów fazy *reduce*, w której następuje obliczenie końcowego wyniku rundy. W podstawowej wersji Hadoopa zarówno dane wejściowe jak i wyjściowe procesów są przechowywane w systemie plików, a dane wymieniane między rundami są zapisywane w HDFS. Prowadzi to do spowolnienia wykonania, ponieważ dostęp do danych na dysku jest dużo wolniejszy niż do danych w pamięci, a w przypadku zapisu do HDFS dane muszą być dodatkowo przesyłane przez sieć. Na Rysunku 2.1 przedstawiono przykład zliczania słów w paradygmacie MapReduce.



Rysunek 2.1: Zliczanie słów w paradygmacie MapReduce

Dodatkowo Hadoop MapReduce zajmuje się monitorowaniem i poprawnym zarządzaniem procesami. W standardowej konfiguracji Hadoopa, Hadoop MapReduce i HDFS działają na tym samym zbiorze węzłów w klastrze. Dzięki temu możliwe jest optymalne przydzielanie procesów do danych na maszynach, co skutkuje zmniejszonym przesyłem danych oraz zwiększeniem wydajności klastra (zamiast na odwrót).

2.1.4. YARN

YARN jest systemem zarządzającym zasobami oraz wykonywaniem się procesów na klastrze. Został dodany w późniejszej wersji Hadoopa (2.0), zastępując Hadoop MapReduce w kwestii zarządzania procesami i znacząco rozszerzając zakres możliwości integracji Hadoopa z innymi modelami programistycznymi. W architekturze klastra można umieścić go pomiędzy HDFS, a serwisami odpowiedzialnymi za uruchamianie aplikacji. Zajmuje się dynamicznym przydzielaniem zasobów aplikacji, optymalizacją ich zużycia oraz uruchamianiem aplikacji.

2.2. Spark

W tym podrozdziale, opierając się na [14], przedstawimy Sparka. Jest on otwartym frameworkiem pozwalającym na szybkie i efektywne przetwarzanie dużych zbiorów danych. Motywacją do stworzenia Sparka były ograniczenia stawiane przez Hadoopa w postaci intesywnego używania dysku oraz niemożliwości wykorzystania danych pośrednich na potrzeby kolejnych operacji. W Hadoopie dane można wykorzystać dopiero po ich wcześniejszym zapisaniu do HDFS. Spark natomiast umożliwia ponowne przetwarzanie cząstkowych danych bez konieczności ich zapisu, a następnie odczytu. Framework działa w oparciu o technologię *in-memory*, umożliwiającą wykonywanie większości obliczeń w pamięci operacyjnej. W skład podstawowego rozwiązania Spark wchodzą:

- SparkCore – zbiór bibliotek (patrz punkt 2.2.1),
- Menedżer zasobów (patrz punkt 2.2.2),
- Rozproszony system danych (patrz punkt 2.2.3),
- RDD (ang. *Resilient Distributed Datasets*) (patrz punkt 2.2.4).

Spark posiada dwa tryby przetwarzania danych: wsadowy i strumieniowy. Znajduje zastosowanie w procesach ETL, analizie danych, uczeniu maszynowym oraz algorytmach grafowych.

2.2.1. SparkCore

SparkCore jest zbiorem bibliotek wspierającym i integrującym pozostałe moduły Sparka. Tworzy warstwę abstrakcji do zarządzania systemem. Zapewnia API wysokiego poziomu dla języków: Scala, Java, Python i R.

2.2.2. Menedżer zasobów

Aplikacja Sparka to niezależne zbiory procesów nadzorowane i zarządzane przez obiekt *SparkContext* tworzony w głównym procesie aplikacji. Na każdym węźle klastra, aplikacji zostaje przydzielony proces wykonawczy odpowiedzialny za jej uruchamianie oraz zarządzanie danymi i zasobami. Systemem zarządzającym zasobami klastra może być Spark, ale również opisany wcześniej Hadoop YARN lub podobny w działaniu Apache Mesos.

2.2.3. Rozproszony system danych

Pomimo rozwiązania *in-memory* czasem zachodzi potrzeba zapisu informacji do trwałego systemu danych. Spark wspiera szeroką gamę rozproszonych systemów przechowywania danych. Na liście znajdują się między innymi: Hadoop HDFS, Amazon S3, Cassandra czy Elasticsearch.

2.2.4. Resilient Distributed Datasets

RDD to podstawowa warstwa abstrakcji danych. Jest to kolekcja obiektów, rozproszona pomiędzy węzły klastra, umożliwiająca bezpieczne i efektywne wykonywanie operacji równoległych. Główne cechy RDD to:

- odporność na błędy (ang. *resilient*) – w przypadku awarii węzła klastra, Spark jest w stanie ponownie obliczyć brakujące lub zniszczone części danych. Odtwarzanie fragmentów danych jest możliwe dzięki przechowywaniu informacji, grafu obliczeń i zależności między danymi (ang. *lineage*),
- rozproszenie danych (ang. *distributed*) – dane kolekcji są trzymane na wielu węzłach klastra jednocześnie,
- kolekcjonowanie danych (ang. *dataset*) – RDD przechowuje obiekty wszelkiego rodzaju, począwszy od typów podstawowych, aż do złożonych rekordów danych,
- zrównoleglenie – obliczenia i przekształcenia wykonywane są w sposób równoległy,
- *in-memory* i utrwalanie danych – RDD jest traktowane jak regularny obiekt, zatem jest przechowywane w pamięci operacyjnej tak długo jak to potrzebne. Istnieje możliwość trwałego zapisu RDD do pamięci podręcznej lub zintegrowanego systemu przechowywania danych, zapewniając tym samym szybszy dostęp do danych pomiędzy procesami,
- leniwa ewaluacja – dane zawarte w RDD nie są dostępne i przetwarzane do momentu wywołania akcji,
- stałość – raz stworzone RDD nie może być zmienione, może być jedynie przekształcone w nowe RDD.

RDD wspierają dwa typy operacji:

- *przekształcenia / transformacje* – leniwie wykonywane obliczenia i operacje zwracające nowe RDD np: *map*, *filter*, *join*,
- *akcje* – obliczenia wykonywane na RDD i zwracające wynik np: *reduce*, *count*.

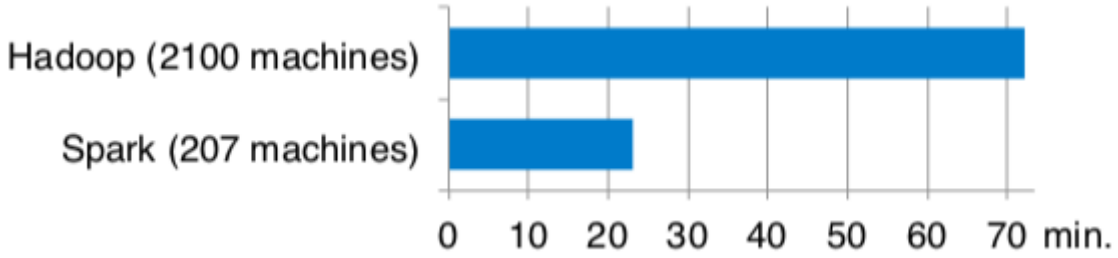
Podstawową jednostką RDD jest partycja, która jest logiczną częścią rozproszonego zbioru danych. Liczbę partycji można kontrolować. Spark optymalizuje przesył danych przez sieć, przez co stara się odtworzyć logiczny podział danych na ich fizyczne odpowiedniki.
Stworzenie RDD umożliwiło efektywne implementowanie:

- interaktywnych narzędzi do analizy danych i wykonywania zapytań,
- iteracyjnych algorytmów używanych w uczeniu maszynowym czy analizie grafów.

2.3. Hadoop vs Spark

Hadoop powstał na trzy lata przed Sparkiem i przez ten czas bardzo dobrze spisywał się jako framework do przetwarzania dużych danych. MapReduce spełniał swoją funkcję jako paradygmat liniowego przetwarzania danych. Jednak z upływem czasu pojawiły się nowe potrzeby, dla których Hadoop nie był dopasowany. Napisanie skomplikowanego algorytmu przy użyciu MapReduce jest czasochłonne. Dodatkowo zaimplementowane rozwiązanie jest zazwyczaj wolne z powodu wielu kroków pośrednich, z których każdy zostaje zapisany do HDFS. W celu usprawnienia powyższych aspektów został stworzony Spark. Umożliwia on pisanie efektywnych i skomplikowanych algorytmów rozproszonych. Używanie grafów wykonań oraz technologii *in-memory* znacząco zredukowało czas wykonania programów. Na Rysunku 2.2 przedstawiono wykres porównujący rezultaty sortowania danych przy użyciu Hadoop MapReduce i Sparka. Program napisany w Sparku posortował dane 3 razy szybciej, używając 10 razy mniej maszyn.

Rysunek 2.2: Porównanie rekordów Hadoopa (2013) i Sparka (2014) w sortowaniu 100 TB danych używając testu wydajności *Daytona Gray* [6].



Dodatkowo Spark udostępnia bardzo rozbudowane API do języków Scala, Java, Python i R. W przeciwieństwie do Hadoopa, tworzenie nowych algorytmów jest intuicyjne, łatwe i szybkie. Oba frameworki posiadają cały szereg zintegrowanych systemów ułatwiających przetwarzanie dużych danych, jednak to Spark jest obecnie bardziej rozwijanym projektem.

Rozdział 3

Algorytmy minimalne

3.1. Definicja

Oznaczmy S jako zbiór obiektów w rozpatrywanym problemie. Niech n będzie liczbą obiektów wchodzących w skład S, a t liczbą maszyn w systemie. Zdefiniujmy $m = n/t$, czyli liczbę obiektów na każdej z maszyn w przypadku równomiernego rozproszenia S. Rozważmy algorytm rozwiązujący pewien problem na zbiorze S. Mówimy, że algorytm jest *minimalny* jeżeli posiada wszystkie z następujących własności [9]:

- *ograniczona pamięć* – każda z maszyn zużywa $O(m)$ pamięci,
- *ograniczony przesył danych* – w każdej rundzie MapReduce / transformacji RDD, każda z maszyn wysyła i odbiera przez sieć co najwyżej $O(m)$ informacji,
- *stała liczba rund* – algorytm musi zakończyć się po stałej liczbie rund / transformacji RDD,
- *optymalność obliczeń* – każda maszyna wykonuje $O(T_{seq}/t)$ obliczeń, gdzie T_{seq} jest czasem potrzebnym na rozwiązanie problemu na pojedynczej maszynie sekwencyjnej. Mianowicie algorytm powinien otrzymać przyspieszenie rzędu t, używając równolegle t maszyn.

Fazy i rundy MapReduce oraz transformacje RDD są w stanie wyrazić dowolne obliczenia rozproszone [15]. Oznacza to również, że obliczenia wykonywane w paradygmacie MapReduce można wyrazić przekształceniami RDD i odwrotnie. Z tego powodu opisy przedstawionych algorytmów minimalnych zostały zapisane tylko w paradygmacie MapReduce.

3.2. TeraSort

TeraSort jest rozproszonym algorytmem sortującym. Na wejściu mamy dany zbiór S składający się z n porównywalnych obiektów. Do dyspozycji mamy t maszyn $M_1, ..., M_t$. Początkowo zbiór S jest równomiernie rozproszony na maszynach. Rezultatem algorytmu jest stan, w którym obiekty na maszynie M_i są posortowane oraz poprzedzają obiekty na maszynach M_j dla każdego $1 \leq i < j \leq t$. Parametryzowany zmienną $\rho \in (0, 1]$ algorytm TeraSort składa się z następujących faz MapReduce [9]:

Runda 1

- *Map-shuffle*
  Na każdej z maszyn M_i $(1 \leq i \leq t)$, odczytujemy z pamięci lokalnej obiekty wejściowe algorytmu i każdy z nich wybieramy z prawdopodobieństwem $\rho = \frac{1}{m}\ln(nt)$ [9]. Na koniec wysyłamy wybrane elementy na wszystkie maszyny $M_1, ..., M_t$.

- *Reduce*
  Niech S' oznacza zbiór elementów otrzymanych z fazy *map*, a $s' = |S'|$. Początkowo sortujemy S', a następnie wybieramy obiekty graniczne $b_1, ..., b_{t-1}$, gdzie b_i jest obiektem o indeksie $i * \lceil s'/t \rceil$ dla $1 \leq i \leq t-1$.

Runda 2

- *Map-shuffle*
  Każda maszyna M_i odczytuje z pamięci lokalnej początkowe obiekty i wysyła elementy należące do przedziału $(b_{j-1}, b_j]$ na maszynę M_j, dla każdego $1 \leq j \leq t$, gdzie $b_0 = -\infty$ oraz $b_t = \infty$.

- *Reduce*
  Każda maszyna M_i sortuje obiekty otrzymane z fazy *map*.

3.3. Lista rankingowa

Niech S oznacza zbiór n porównywalnych obiektów wejściowych. Rezultatem algorytmu jest zwrócenie pozycji rankingowej $rank(o) = |\{o' \in S : \ o' \leq o\}|$, dla każdego $o \in S$. Problem można rozwiązać w czasie *O(n log n)* na pojedynczej maszynie. Przykładowo dla $S = \{4, 8, 1, 7, 10\}$ lista rankingowa prezentuje się następująco:

$$rank(1) = 1$$
$$rank(4) = 2$$
$$rank(7) = 3$$
$$rank(8) = 4$$
$$rank(10) = 5$$

Poniżej została przedstawiona rozproszona wersja algorytmu w oparciu o paradygmat MapReduce [9].

Rundy 1 – 2
Sortujemy S przy użyciu algorytmu $TeraSort$.

Runda 3
Niech $Sorted_i$ oznacza posortowany zbiór obiektów na maszynie M_i, dla $1 \leq i \leq t$.

- *Map-shuffle*
  Każda maszyna M_i wysyła $|Sorted_i|$ na maszyny $M_{i+1}, ..., M_t$.

- *Reduce*
  Niech:
  - $R_i = \sum_{j \leq i-1} |Sorted_j|$
  - $localRank(o) = |\{o' \in Sorted_i : \ o' \leq o\}|$ dla każdego $o \in Sorted_i$

  Numer rankingowy wynosi:

$$rank(o) = R_i + localRank(o)$$

3.4. Statystyki prefiksowe

Niech S oznacza zbiór n porównywalnych obiektów wejściowych, a *stat* będzie funkcją statystyk łączną na zbiorze S. Z łączności wynika zatem, że $stat(S)$ może zostać obliczone w czasie stałym ze $stat(S_1)$ i $stat(S_2)$, gdzie S_1 i S_2 tworzą podział S, czyli $S_1 \cup S_2 = S$ oraz $S_1 \cap S_2 = \emptyset$. Rezultat algorytmu definiujemy następująco [9]:

$$prefixStat(o, stat) = stat(\{o' \in S : \ o' < o\}) \text{ dla każdego } o \in S$$

Przykładową funkcją *stat* jest *suma*. Niech $S = \{1, 2, 3, 4\}$, wtedy

$$\begin{aligned} prefixStat(1, suma) &= suma(\emptyset) = 0 \\ prefixStat(2, suma) &= suma(\{1\}) = 1 \\ prefixStat(3, suma) &= suma(\{1, 2\}) = 1 + 2 = 3 \\ prefixStat(4, suma) &= suma(\{1, 2, 3\}) = 1 + 2 + 3 = 6 \end{aligned}$$

Rundy 1 – 2
Sortujemy S przy użyciu algorytmu $TeraSort$.

Runda 3
Niech $Sorted_i$ oznacza posortowany zbiór obiektów na maszynie M_i, dla $1 \leq i \leq t$.

- *Map-shuffle*
  Każda maszyna M_i wysyła $stat(Sorted_i)$ na maszyny $M_{i+1}, ..., M_t$.
- *Reduce*
  Niech
  - $V_i = stat(\{stat(Sorted_1), stat(Sorted_2), ..., stat(Sorted_{i-1})\})$
  - $prefixLocal(o, stat) = stat(\{o' \in Sorted_i : \ o' < o\})$ dla każdego $o \in Sorted_i$

  Statystyki prefiksowe wynoszą:

$$prefixStat(o, stat) = V_i + prefixLocal(o, stat) \text{ dla każdego } o \in Sorted_i$$

3.5. Grupowanie

Niech S oznacza zbiór n porównywalnych obiektów wejściowych, a *stat* będzie funkcją statystyk łączną na zbiorze S. Dodatkowo dla każdego $o \in S$ istnieje funkcja $key(o)$ zwracająca obiekt będący porównywalnym kluczem danego elementu o. Grupą G nazwiemy maksymalny zbiór obiektów, dla których funkcja key zwraca tą samą wartość.

Rezultat algorytmu *grupowania* definiujemy następująco [9]:

$$groupBy(G, stat) = stat(G) \text{ dla każdej grupy } G \text{ na zbiorze } S$$

Przykładowo niech funkcją *stat* będzie *suma*, $S = \{1, 2, 3, 4, 5\}$ oraz

$$key(1) = 42$$
$$key(2) = 42$$
$$key(3) = 35$$
$$key(4) = 35$$
$$key(5) = 42$$

Następnie wyznaczamy grupy:

$$G_{42} = \{1, 2, 5\}$$
$$G_{35} = \{3, 4\}$$

Końcowy wynik algorytmu wynosi:

$$groupBy(G_{42}, suma) = suma(\{1, 2, 5\}) = 8$$
$$groupBy(G_{35}, suma) = suma(\{3, 4\}) = 7$$

Rundy 1 – 2
Sortujemy S przy użyciu algorytmu $TeraSort$.

Runda 3
Niech $Sorted_i$ oznacza posortowany zbiór obiektów na maszynie M_i, dla $1 \leq i \leq t$.

- *Map-shuffle*
  Niech:
  - $k_{min}(i) = min(\{key(o) : \ o \in Sorted_i\})$
  - $k_{max}(i) = max(\{key(o) : \ o \in Sorted_i\})$

  Jeżeli dla danego klucza k, grupa G_k znajduje się całkowicie na maszynie M_i, to $groupBy(G_k, stat)$ obliczamy lokalnie na M_i, a następnie przesyłamy parę $(k, groupBy(G_k, stat))$ na maszynę M_i. Takie grupy są tworzone przez obiekty $o \in Sorted_i$, takie że $k_{min}(i) < key(o) < k_{max}(i)$. Wynika z tego, że każda maszyna M_i ma co najwyżej dwie grupy, które nie mogą zostać obliczone lokalnie:

  $$G_{min}(i) = \{o : \ o \in Sorted_i \wedge key(o) = k_{min}(i)\}$$
  $$G_{max}(i) = \{o : \ o \in Sorted_i \wedge key(o) = k_{max}(i)\}$$

  Stąd wiadomo, że takich grup na wszystkich maszynach jest maksymalnie $2t$. Aby obliczyć wynik dla grup niemieszczących się na pojedynczej maszynie, najpierw obliczamy $stat(G_{min}(i))$ oraz $stat(G_{max}(i))$ lokalnie na M_i. Następnie wybieramy pojedynczą maszynę, np. M_1, na której zgrupujemy lokalne wyniki. W ostatnim kroku wysyłamy parę $(k_{min}(i), stat(G_{min}(i)))$ na wybraną maszynę M_1 oraz jeżeli $k_{min}(i) \neq k_{max}(i)$, to wysyłamy również parę $(k_{max}(i), stat(G_{max}(i)))$ na maszynę M_1. Widzimy, że policzenie statystyk na takich grupach wymaga przesłania dwóch lub czterech wartości na pojedynczą maszynę, na której zostanie obliczony końcowy wynik.

- *Reduce*
  Na maszynach $M_2, ..., M_t$ mamy już gotowe wyniki grupowania. Natomiast na wybranej maszynie M_1 niech $(k_1, w_1), ..., (k_x, w_x)$ oznaczają pary otrzymane z fazy *map*, gdzie $x \in [t, 2t]$. Dla każdej grupy, której klucz k jest wśród otrzymanych par, wynikiem algorytmu jest:

$$G_k = stat(\{w_j : \ k = k_j\})$$

3.6. Pół-złączenia

Niech R i T będą dwoma zbiorami z tej samej dziedziny. Dla każdego obiektu $o \in R \cup T$ istnieje funkcja $key(o)$ zwracająca klucz obiektu o. Problem *pół-złączeń* polega na znalezieniu wszystkich obiektów $o \in R$, takich że istnieje obiekt $o' \in T$, dla którego $key(o) = key(o')$. Zakładamy również, że elementy zbiorów są zdefiniowane w sposób pozwalający na rozróżnienie, z którego zbioru pochodzą. Problem pół-złączeń posiada rozwiązanie o złożoności czasowej $O(n \ log \ n)$ na pojedynczej maszynie sekwencyjnej, gdzie $n = |R \cup T|$ [9].

Przykładowo niech $R = \{1, 2, 5\}$, $T = \{7, 8, 9\}$ oraz

$$\begin{array}{ll} key(1) = 20 & key(7) = 21 \\ key(2) = 21 & key(8) = 23 \\ key(5) = 22 & key(9) = 20 \end{array}$$

Wynikiem algorytmu jest zbiór $\{1, 2\}$, ponieważ $key(1) = key(9)$ oraz $key(2) = key(7)$.

Rundy 1 – 2
Sortujemy $R \cup T$ przy użyciu algorytmu $TeraSort$.

Runda 3
Zdefiniujmy R_i oraz T_i jako zbiory obiektów znajdujących się na maszynie M_i i początkowo należących odpowiednio do zbiorów R, T.

- *Map-shuffle*
  Na każdej maszynie M_i, dla $1 \leq i \leq t$, wysyłamy do wszystkich maszyn następujące dwie wartości:
  - $min(\{key(o) : o \in T_i\})$,
  - $max(\{key(o) : o \in T_i\})$.
- *Reduce*
  Niech T_{border} będzie zbiorem kluczy otrzymanych z fazy *map*. Na każdej z maszyn M_i, dla $1 \leq i \leq t$, zwracamy obiekt $o \in R_i$ jako część rezultatu, gdy

$$key(o) \in T_i \cup T_{border}$$

3.7. Sortowanie z perfekcyjnym zrównoważeniem

Niech S będzie zbiorem składającym się z n porównywalnych obiektów, a $m = \lceil n/t \rceil$. Aby otrzymać rezultat algorytmu, należy przegrupować elementy, tak aby każda z maszyn $M_1, ..., M_{t-1}$ zawierała dokładnie m obiektów, a maszyna M_t zawierała pozostałe. Dodatkowo obiekty na maszynie M_i, dla $1 \leq i \leq t$, są posortowane oraz poprzedzają obiekty na maszynach M_j dla $1 \leq i < j \leq t$.

Rundy 1 – 3
Na zbiorze S wykonujemy algorytm *listy rankingowej*. Niech $rank(o)$ dla $o \in S$ oznacza pozycję rankingową obiektu o.

Runda 4
Wykonujemy tylko fazę *map-shuffle*, w której dla każdego obiektu o na maszynie M_i, dla $1 \leq i \leq t$, wysyłamy go na maszynę M_j, gdzie $j = \lceil rank(o)/m \rceil$.

3.8. Statystyka okienkowa

Niech:

- S – zbiór n porównywalnych obiektów,
- l – wielkość okna statystyk, $l \leq n$,
- $stat$ – funkcja statystyk łączna na zbiorze S.

Dla każdego $o \in S$ zdefiniujmy $window(o)$ jako zbiór l największych obiektów nie przekraczających o. Statystyka okienkowa obiektu o wynosi:

$$winStat(o) = stat(window(o))$$

Rezultatem algorytmu *statystyki okienkowej* [9] jest zwrócenie wartości $winStat(o)$ dla każdego obiektu $o \in S$.

Przykładowo dla $S = \{1, 2, 3, 4, 5\}$, $l = 3$ oraz $stat = suma$ wynikiem algorytmu jest:

$$\begin{aligned} winStat(1) &= suma(\{1\}) = 1 \\ winStat(2) &= suma(\{1, 2\}) = 3 \\ winStat(3) &= suma(\{1, 2, 3\}) = 6 \\ winStat(4) &= suma(\{2, 3, 4\}) = 9 \\ winStat(5) &= suma(\{3, 4, 5\}) = 12 \end{aligned}$$

Rundy 1 – 4
Wykonujemy algorytm *sortowania z perfekcyjnym zrównoważeniem*. W dalszej części algorytmu $m = \lceil n/t \rceil$.

Runda 5
Niech S_i oznacza zbiór elementów na maszynie M_i, dla $1 \leq i \leq t$.

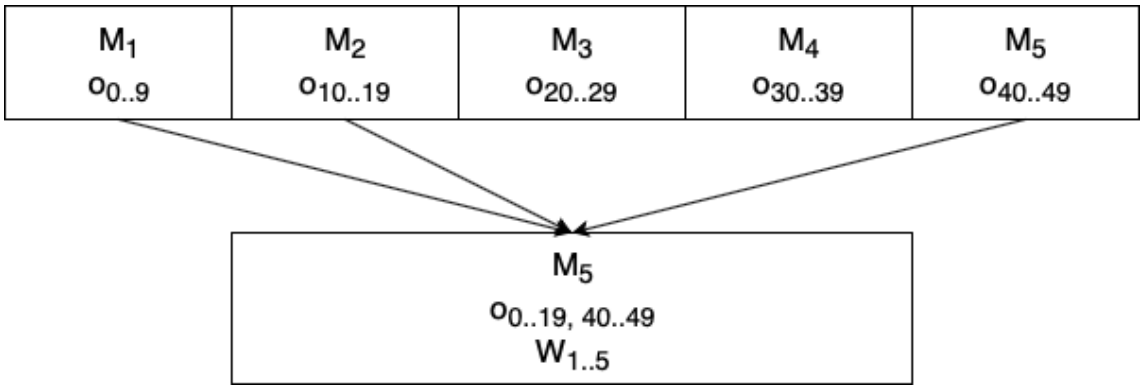
- *Map-shuffle*
  Do wszystkich maszyn wysyłamy $W_i = stat(S_i)$. Następnie, zgodnie z lematem 2 z pracy [9], każdy obiekt $o \in S_i$ musi zostać wysłany na maksymalnie dwie dodatkowe maszyny. Zatem przesyłamy wszystkie obiekty zbioru S_i na maszynę M_{i+1}, jeżeli $l \leq m$, a w przeciwnym przypadku na maszyny o indeksach $i+\lfloor(l-1)/m\rfloor$ oraz $i+1+\lfloor(l-1)/m\rfloor$.

- *Reduce*
  Dla każdego $o \in S_i$ obliczamy:
  - $\alpha = \lceil(rank(o) - l + 1)/m\rceil$,
  - $w_1 = stat(\{o' : o' \in S_\alpha \wedge o' \in window(o)\})$. Jeżeli $\alpha < i$, to takie obiekty o' zostały przysłane na maszynę M_i w fazie *map-shuffle* rundy 5,
  - $w_2 = \sum_{j=\alpha+1}^{i-1} W_j$,
  - jeżeli $\alpha = i$, to $w_3 = 0$, wpp $w_3 = stat(\{o' : o' \in S_i \wedge o' \in window(o)\})$.

$$winStat(o) = stat(\{w_1, w_2, w_3\})$$

Na Rysunku 3.1 przedstawiono przykład działania rundy 5 dla maszyny $i = 5$. Niech o_r oznacza obiekt $o \in S$ z pozycją rankingową równą r oraz $l = 35, m = 10$.

Rysunek 3.1: Rozesłanie obiektów podczas rundy 5 algorytmu agregacji okienkowej



Wartość agregacji okienkowej dla o_{42} obliczamy następująco:

$$\begin{aligned} \alpha &= 1 \\ w_1 &= stat(\{o_8, o_9\}) \\ w_2 &= stat(\{W_2, W_3, W_4\}) \\ w_3 &= stat(\{o_{40}, o_{41}, o_{42}\}) \\ winStat(o_{42}) &= stat(\{w_1, w_2, w_3\} \end{aligned}$$

Rozdział 4

Biblioteka do tworzenia algorytmów minimalnych

Z rozdziału 3 wynika jak dużo wspólnych obliczeń i funkcjonalności posiadają poszczególne algorytmy minimalne. Należą do nich między innymi sortowanie obiektów, liczenie łącznych statystyk na zbiorach, wysyłanie obiektów na wybrane maszyny oraz równolegle przesyłanie wielu typów obiektów. W poniższym rozdziale przedstawię implementację mojej biblioteki, napotkane problemy i ich rozwiązania. Podrozdziały Hadoop 4.2 oraz Spark 4.3 przedstawiają kolejno:

1. Implementację algorytmu minimalnego bez użycia biblioteki,
2. Implementację algorytmu minimalnego z użyciem biblioteki,
3. Opis implementacji biblioteki.

4.1. Wstęp

MinimalAlgFactory jest klasą udostępniającą zaimplementowane algorytmy minimalne. Została napisana w języku Java 8. Jej zadaniem jest połączenie implementacji algorytmów minimalnych na platformy Hadoop i Spark. Dzięki takiemu rozwiązaniu użytkownik może raz napisany algorytm uruchamiać na jednym z frameworków bez konieczności dostosowywania. W skład klasy wchodzą algorytmy:

- *TeraSort* – sortowanie,
- *Rank* – lista rankingowa,
- *PerfectSort* – sortowanie z perfekcyjnym równoważeniem,
- *Prefix* – statystyki prefiksowe,
- *SemiJoin* – pół-złączenia,
- *GroupBy* – grupowanie,
- *SlidingAggregation* – statystyka okienkowa.

4.2. Hadoop

4.2.1. Implementacja algorytmu bez użycia biblioteki

Listingi 4.1 – 4.9 przedstawiają implementację algorytmu statystyk prefiksowych przy użyciu podstawowej wersji Hadoopa 2.8. Funkcją statystyk jest suma. Na Listingu 4.1 ukazano klasę obiektów wejściowych. W plikach z danymi każda linia składa się z czterech liczb całkowitych oddzielonych spacją i reprezentuje pojedynczy obiekt klasy $FourInts$. W liniach 22 – 24 została zaimplementowana funkcja obliczającą wartość obiektu, która będzie wykorzystywana w liczeniu sum prefiksowych. Natomiast linie 35 – 42 przedstawiają komparator użyty do porównywania obiektów $FourInts$.

Listing 4.1: Klasa danych wejściowych do algorytmu statystyk prefiksowych

```
1  public class FourInts {
2    public static final String DELIMITER = " ";
3    private int first;
4    private int second;
5    private int third;
6    private int fourth;
7    private String encodedValues;
8
9    public FourInts(String encodedValues) {
10     this.encodedValues = encodedValues;
11     String[] values = encodedValues.split(DELIMITER);
12     this.first = Integer.parseInt(values[0]);
13     this.second = Integer.parseInt(values[1]);
14     this.third = Integer.parseInt(values[2]);
15     this.fourth = Integer.parseInt(values[3]);
16   }
17
18   public FourInts(Text encodedValues) {
19     this(encodedValues.toString());
20   }
21
22   public long getValue() {
23     return this.second + this.third + this.fourth;
24   }
25
26   @Override
27   public String toString() {
28     return this.encodedValues;
29   }
30
31   public Text toText() {
32     return new Text(this.toString());
33   }
34
35   public static class FourIntsComparator implements Comparator<FourInts> {
36     @Override
37     public int compare(FourInts o1, FourInts o2) {
38       return o1.first > o2.first ? 1 : (o1.first < o2.first ? -1 : 0);
39     }
40   }
41
42   public static Comparator<FourInts> cmp = new FourIntsComparator();
43 }
```

Klasa *MultipleFourInts* z Listingu 4.2 przedstawia prosty serializator kolekcji obiektów *FourInts*. Zostanie wykorzystana przy implementacji rund algorytmu na Listingach 4.6 – 4.9.

Listing 4.2: Kolekcja obiektów FourInts

```
public class MultipleFourInts {
  public static final String DELIMITER = "#";
  private List<FourInts> fourIntsSeq;

  public MultipleFourInts(Text encodedValues){
    this(encodedValues.toString());
  }

  public MultipleFourInts(String encodedValues) {
    this.fourIntsSeq = new ArrayList<>();
    for (String value : encodedValues.split(DELIMITER)) {
      this.fourIntsSeq.add(new FourInts(value));
    }
  }

  public MultipleFourInts(List<FourInts> fourIntsSeq) {
    this.fourIntsSeq = fourIntsSeq;
  }

  public List<FourInts> getValues() {
    return this.fourIntsSeq;
  }

  @Override
  public String toString() {
    List<String> result = new ArrayList();
    for (FourInts fourInts : this.fourIntsSeq) {
      result.add(fourInts.toString());
    }
    return String.join(DELIMITER, result);
  }

  public Text toText() {
    return new Text(this.toString());
  }
}
```

Listing 4.3 przedstawia klasę, w której trzymane są statystyki dla całych maszyn. Atrybut *key* oznacza numer maszyny, z której pochodzą statystyki, a atrybut *statistics* oznacza wartość sumy obiektów na danej maszynie. W liniach 21 – 33 zdefiniowano komparator, umożliwiający posortowanie statystyk względem numerów maszyn.

Listing 4.3: Klasa indeksowanych statystyk

```
1  public class IndexedStatistics {
2    public static final String DELIMITER = "%";
3    public int key;
4    public long statistics;
5    public String encodedValue;
6
7    public IndexedStatistics(String key, long statistics) {
8      this.encodedValue = key + DELIMITER + Long.toString(statistics);
9    }
10
11   public IndexedStatistics(Text encodedValue) {
12     String[] tokens = encodedValue.toString().split(DELIMITER);
13     this.key = Integer.parseInt(tokens[0]);
14     this.statistics = Long.parseLong(tokens[1]);
15   }
16
17   public Text toText() {
18     return new Text(this.encodedValue);
19   }
20
21   public static class IndexedStatisticsComparator
22         implements Comparator<IndexedStatistics> {
23     @Override
24     public int compare(IndexedStatistics o1, IndexedStatistics o2) {
25       return o1.key > o2.key ?
26         1 : (o1.key < o2.key ?
27           -1 : (o1.statistics > o2.statistics ?
28             1 : o1.statistics < o2.statistics ? -1 : 0));
29     }
30   }
31
32   public static Comparator<IndexedStatistics> cmp =
33     new IndexedStatisticsComparator();
34 }
```

Na Listingu 4.4 widzimy klasę uruchamiającą algorytm. W liniach 17 – 24 przygotowujemy konfigurację programu, dostępną przez cały czas wykonania programu, w każdej fazie *map* i *reduce*. Jest to standardowy sposób udostępniania niewielkich wartości wszystkim maszynom. Następnie w liniach 27 – 31 uruchamiamy kolejne rundy algorytmu. Widzimy, że danymi wejściowymi do kolejnych rund są wyniki poprzednich.

Listing 4.4: Uruchamianie algorytmu

```
1  public class PrefixApp extends Configured implements Tool {
2    private int computeRatioRandom(int itemsNo, int partitionsNo) {
3      int itemsPerPartition = 1 + itemsNo / partitionsNo;
4      double rho = 1. / itemsPerPartition *
5        Math.log(((double) itemsNo) * partitionsNo);
6      return (int) (1 / rho);
7    }
8
9    public int run(String[] args) throws Exception {
10     if (args.length != 6) {
11       System.err.println("Usage: PrefixApp" +
12         "<home_dir> <input_path> <output_path>" +
13         "<items_no> <partitions_no> <reduce_tasks_no>");
14       return -1;
15     }
16
17     Configuration conf = getConf();
18     int itemsNo = Integer.parseInt(args[3]);
19     int partitionsNo = Integer.parseInt(args[4]);
20     int reducersNo = Integer.parseInt(args[5]);
21
22     conf.setInt(Utils.RATIO_KEY, computeRatioRandom(itemsNo, partitionsNo));
23     conf.setInt(Utils.SPLIT_POINTS_KEY, partitionsNo);
24     conf.setInt(Utils.REDUCERS_NO_KEY, reducersNo);
25
26     Path input = new Path(args[1]);
27     Sampling.run(input, new Path("samp"), conf);
28     Sorting.run(input, new Path(args[0] + "/samp"), new Path("sort"), conf);
29     PartStats.run(new Path(args[0] + "/sort"), new Path("part_stats"), conf);
30     return Prefix.run(new Path(args[0] + "/sort"),
31         new Path(args[0] + "/part_stats"), new Path("prefix"), conf);
32   }
33
34   public static void main(String[] args) throws Exception {
35     int res = ToolRunner.run(new Configuration(), new PrefixApp(), args);
36     System.exit(res);
37   }
38 }
```

Przed wykonaniem faz *map* i *reduce* Hadoop MapReduce umożliwia zapisanie danych w postaci tekstowej do lokalnego systemu plików każdego *DataNode* – funkcja *addCacheFile*. Klasa *Utils* z Listingu 4.5 udostępnia pomocniczą metodę odczytywania obiektów z tak zapisanych plików. Pozwala to na szybki dostęp do niedużych obiektów niezbędnych na każdej maszynie.

Listing 4.5: Metody pomocnicze

```
1  public class Utils {
2    public static final String RATIO_KEY = "ratio";
3    public static final String SPLIT_POINTS_KEY = "splitPoints";
4    public static final String REDUCERS_NO_KEY = "reducersNo";
5
6    public static ArrayList<String> readFromCache(Path filePath) {
7      ArrayList<String> words = new ArrayList<>();
8      try {
9        BufferedReader bufferedReader = new BufferedReader(
10         new FileReader(filePath.toString()));
11       String word = null;
12       while((word = bufferedReader.readLine()) != null) {
13         words.add(word);
14       }
15     } catch(IOException ex) {
16       System.err.println(ex.getMessage());
17     }
18     return words;
19   }
20 }
```

Na Listingu 4.6 przedstawiono pierwszą rundę algorytmu *TeraSort*. W linii 10 odczytujemy z konfiguracji, uzupełnionej przy uruchomieniu programu, wartość prawdopodobieństwa ρ zdefiniowanego w podrozdziale 3.2. Następnie w liniach 16 – 18 wybieramy obiekty wejściowe z prawdopodobieństwem ρ i wysyłamy na jedną z maszyn. Warto zauważyć, że w celu wysłania wszystkich wybranych obiektów na jedną maszynę używamy klucza o wartości NULL.

W fazie *reduce* odbieramy wybrane obiekty i parsujemy do typu *FourInts*. Następnie sortujemy i wyznaczamy miejsca podziałów. Warto zauważyć, że w linii 53 ustawiamy ilość reducerów na 1, dzięki czemu wszystkie miejsca podziałow znajdą się w jednym pliku.

Listing 4.6: Faza próbkowania danych

```
1  public class Sampling {
2    public static class SamplerMapper extends
3          Mapper<LongWritable, Text, NullWritable, Text> {
4      private final Random random = new Random();
5      private int ratioForRandom;
6
7      @Override
8      public void setup(Context ctx) throws IOException, InterruptedException {
9        super.setup(ctx);
10       ratioForRandom = ctx.getConfiguration().getInt(Utils.RATIO_KEY, -1);
11     }
12
13     @Override
14     public void map(LongWritable key, Text value, Context context) throws
15         IOException, InterruptedException {
16       if (random.nextInt(ratioForRandom) == 0) {
17         context.write(NullWritable.get(), value);
18       }
```

```
    }
  }

  public static class ComputeBoundsForSortingReducer extends
        Reducer<NullWritable, Text, Text, NullWritable> {
    private int noOfSplitPoints;

    @Override
    public void setup(Context ctx) throws IOException, InterruptedException {
      super.setup(ctx);
      Configuration conf = ctx.getConfiguration();
      noOfSplitPoints = conf.getInt(Utils.SPLIT_POINTS_KEY, 0) - 1;
    }

    @Override
    protected void reduce(NullWritable key, Iterable<Text> values,
        Context context) throws IOException, InterruptedException {
      ArrayList<FourInts> result = new ArrayList<>();
      for (Text value : values) {
        result.add(new FourInts(value));
      }

      java.util.Collections.sort(result, FourInts.cmp);
      int step = result.size() / (noOfSplitPoints+1);
      for (int i = 1; i <= noOfSplitPoints; i++) {
        context.write(result.get(i * step).toText(), NullWritable.get());
      }
    }
  }

  public static int run(Path input, Path output, Configuration conf) throws
        IOException, InterruptedException, ClassNotFoundException {
    Job job = Job.getInstance(conf, "JOB: Phase sampling");
    job.setJarByClass(Sampling.class);
    job.setNumReduceTasks(1);

    FileInputFormat.setInputPaths(job, input);
    FileOutputFormat.setOutputPath(job, output);

    job.setMapperClass(SamplerMapper.class);
    job.setReducerClass(ComputeBoundsForSortingReducer.class);

    job.setMapOutputKeyClass(NullWritable.class);
    job.setMapOutputValueClass(Text.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(NullWritable.class);

    return job.waitForCompletion(true) ? 0 : 1;
  }
}
```

Listing 4.7 przedstawia rundę 2 algorytmu *TeraSort*. Ważnym momentem są linie 46 – 47, w których zapisujemy do pamięci lokalnej każdej z maszyn punkty podziału obliczone w poprzedniej rundzie – *Próbkowanie*, Listing 4.6. W fazie *map* dla każdego obiektu wejściowego obliczamy numer przedziału, do którego należy. Zauważmy, że obliczony numer będzie wysłany jako klucz, czyli wszystkie obiekty należące do danego przedziału zostaną przetworzone przez ten sam proces *reduce*. W fazie *reduce* sortujemy obiekty należące do danego przedziału i zapisujemy w postaci *<numer przedziału, posortowana lista obiektów>*. W ten sposób symulujemy przypisanie obiektów do maszyn opisywane w rozdziale 3. Posortowana lista obiektów zostaje zapisana w obiekcie klasy *MultipleFourInts*.

Listing 4.7: Faza sortowania danych

```
1  public class Sorting {
2    public static final String SAMPLING_SPLIT_POINTS_CACHE =
3      "sampling_split_points.cache";
4
5    public static class PartitioningMapper extends
6          Mapper<LongWritable, Text, IntWritable, Text> {
7      private FourInts[] splitPoints;
8
9      @Override
10     public void setup(Context ctx) {
11       ArrayList<String> words = Utils.readFromCache(
12         new Path(SAMPLING_SPLIT_POINTS_CACHE));
13       splitPoints = new FourInts[words.size()];
14       for (int i = 0; i < words.size(); i++) {
15         splitPoints[i] = new FourInts(words.get(i));
16       }
17     }
18
19     @Override
20     protected void map(LongWritable key, Text value, Context context) throws
21         IOException, InterruptedException {
22       int dummy = java.util.Arrays.binarySearch(splitPoints,
23         new FourInts(value), FourInts.cmp);
24       context.write(new IntWritable(dummy >= 0 ? dummy : -dummy - 1), value);
25     }
26   }
27
28   public static class SortingReducer extends
29         Reducer<IntWritable, Text, IntWritable, Text> {
30     @Override
31     protected void reduce(IntWritable key, Iterable<Text> values,
32         Context context) throws IOException, InterruptedException {
33       List<FourInts> result = new ArrayList<>();
34       for (Text record : values) {
35         result.add(new FourInts(record));
36       }
37       java.util.Collections.sort(result, FourInts.cmp);
38       context.write(key, new MultipleFourInts(result).toText());
39     }
40   }
41
42   public static int run(Path input, Path samplingSuperdir, Path output,
43         Configuration conf) throws Exception {
44     Job job = Job.getInstance(conf, "JOB: Phase Sorting Reducer");
45     job.setJarByClass(Sorting.class);
46     job.addCacheFile(new URI(samplingSuperdir + "/part-r-00000" + "#" +
47       Sorting.SAMPLING_SPLIT_POINTS_CACHE));
```

```
48      job.setNumReduceTasks(conf.getInt(Utils.REDUCERS_NO_KEY, 1));
49      job.setMapperClass(PartitioningMapper.class);
50      job.setMapOutputKeyClass(IntWritable.class);
51      job.setMapOutputValueClass(Text.class);
52
53      FileInputFormat.setInputPaths(job, input);
54      FileOutputFormat.setOutputPath(job, output);
55
56      job.setReducerClass(SortingReducer.class);
57      job.setOutputKeyClass(IntWritable.class);
58      job.setOutputValueClass(Text.class);
59
60      return job.waitForCompletion(true) ? 0 : 1;
61    }
62  }
```

Na Listingach 4.8 oraz 4.9 przedstawiono rundę 3 algorytmu *statystyki prefiksowe*. Rozbito ją na dwie faktyczne rundy, ponieważ podstawowa wersja Hadoopa nie posiada łatwego sposobu przesyłania różnych typów obiektów między fazami *map* i *reduce*. Taka potrzeba zachodzi w momencie przesyłania zarówno statystyk dla całych maszyn, obiekty *IndexedStatistics*, jak i obiektów klasy *FourInts*, dla których mają zostać policzone statystyki prefiksowe. Klasa *PartStats* z Listingu 4.8 zastępuje fazę $map - shuffle$ z rundy 3 algorytmu *statystyk prefiksowych* i oblicza statystyki prefiksowe dla całych maszyn, a następnie zapisuje je jako pojedynczy plik w HDFS.

Listing 4.8: Faza liczenia statystyk prefiksowych dla całych maszyn

```
1   public class PartStats {
2     public static class PartPrefixMapper extends
3           Mapper<Text, Text, NullWritable, Text> {
4       @Override
5       protected void map(Text key, Text value, Context context) throws
6           IOException, InterruptedException {
7         long result = 0;
8         for (FourInts fourInts : new MultipleFourInts(value).getValues()) {
9           result += fourInts.getValue();
10        }
11        context.write(NullWritable.get(),
12                      new IndexedStatistics(key.toString(), result).toText());
13      }
14    }
15
16    public static class PartStatsReducer extends
17          Reducer<NullWritable, Text, NullWritable, LongWritable> {
18      @Override
19      protected void reduce(NullWritable key, Iterable<Text> values,
20          Context context) throws IOException, InterruptedException {
21        List<IndexedStatistics> result = new ArrayList<>();
22        for (Text value : values) {
23          result.add(new IndexedStatistics(value));
24        }
25        java.util.Collections.sort(result, IndexedStatistics.cmp);
26
27        Long[] prefixPartStats = new Long[result.size()];
28        for (int i = 0; i < result.size(); i++) {
29          long stat = i == 0 ?
30              0 : result.get(i-1).statistics + prefixPartStats[i-1];
31          prefixPartStats[i] = stat;
32          context.write(NullWritable.get(), new LongWritable(stat));
```

```
33        }
34      }
35    }
36
37    public static int run(Path input, Path output, Configuration conf) throws
38          Exception {
39      Job job = Job.getInstance(conf, "JOB: Phase PartStats");
40      job.setJarByClass(PartStats.class);
41      job.setNumReduceTasks(1);
42      job.setMapperClass(PartPrefixMapper.class);
43      job.setInputFormatClass(KeyValueTextInputFormat.class);
44      job.setMapOutputKeyClass(NullWritable.class);
45      job.setMapOutputValueClass(Text.class);
46
47      FileInputFormat.setInputPaths(job, input);
48      FileOutputFormat.setOutputPath(job, output);
49
50      job.setReducerClass(PartStatsReducer.class);
51      job.setOutputKeyClass(NullWritable.class);
52      job.setOutputValueClass(LongWritable.class);
53
54      return job.waitForCompletion(true) ? 0 : 1;
55    }
56  }
```

Danymi wejściowymi do programu z Listingu 4.9 jest wynik fazy sortowania. W liniach 59 – 60 statystyki prefiksowe całych maszyn obliczone na Listingu 4.8 są zapisywane na pamięć lokalną każdej z maszyn. Faza *map* tylko transferuje obiekty do fazy *reduce*. W liniach 18 – 21 odczytujemy z pamięci lokalnej i parsujemy statystyki prefiksowe maszyn. Ostatecznie w metodzie *reduce* obliczamy końcowy wynik dla każdego obiektu wejściowego.

Listing 4.9: Ostatnia faza liczenia statystyk prefiksowych

```
1  public class Prefix {
2    static final String PREFIX_PART_STATS_CACHE = "prefix_part_stats.cache";
3
4    public static class PrefixMapper extends Mapper<Text, Text, Text, Text> {
5      @Override
6      protected void map(Text key, Text value, Context context) throws
7          IOException, InterruptedException {
8        context.write(key, value);
9      }
10   }
11
12   public static class PrefixReducer extends
13         Reducer<Text, Text, LongWritable, Text> {
14     List<Long> machinePrefixStats;
15
16     @Override
17     public void setup(Context ctx) {
18       ArrayList<String> words = Utils.readFromCache(
19         new Path(PREFIX_PART_STATS_CACHE));
20       machinePrefixStats = words.stream().
21         map(w -> Long.parseLong(w)).collect(Collectors.toList());
22     }
23
24
25     @Override
26     protected void reduce(Text key, Iterable<Text> values, Context context)
27         throws IOException, InterruptedException {
```

```
28        List<FourInts> fourIntsList = new ArrayList<>();
29        for (Text value : values) {
30          for (FourInts fourInts : new MultipleFourInts(value).getValues()) {
31            fourIntsList.add(fourInts);
32          }
33        }
34        java.util.Collections.sort(fourIntsList, FourInts.cmp);
35
36        int partIndex = Integer.parseInt(key.toString());
37        long prevPartStats = machinePrefixStats.get(partIndex);
38
39        Long[] result = new Long[fourIntsList.size()];
40        for (int i = 0; i < fourIntsList.size(); i++) {
41          long stat = i == 0 ?
42            prevPartStats : fourIntsList.get(i-1).getValue() + result[i-1];
43          result[i] = stat;
44          context.write(new LongWritable(stat), fourIntsList.get(i).toText());
45        }
46      }
47    }
48
49    public static int run(Path input, Path samplingSuperdir, Path output,
50          Configuration conf) throws Exception {
51      Job job = Job.getInstance(conf, "JOB: Phase Prefix");
52      job.setJarByClass(Prefix.class);
53      job.setNumReduceTasks(conf.getInt(Utils.REDUCERS_NO_KEY, 1));
54      job.setMapperClass(PrefixMapper.class);
55      job.setInputFormatClass(KeyValueTextInputFormat.class);
56      job.setMapOutputKeyClass(Text.class);
57      job.setMapOutputValueClass(Text.class);
58
59      job.addCacheFile(new URI(samplingSuperdir + "/part-r-00000" + "#" +
60        PREFIX_PART_STATS_CACHE));
61      FileInputFormat.setInputPaths(job, input);
62      FileOutputFormat.setOutputPath(job, output);
63
64      job.setReducerClass(PrefixReducer.class);
65      job.setOutputKeyClass(LongWritable.class);
66      job.setOutputValueClass(Text.class);
67
68      return job.waitForCompletion(true) ? 0 : 1;
69    }
70  }
```

Warto zauważyć, że wszystkie fazy *map* i *reduce* jako dane wejściowe przyjmują Text, IntWritable lub LongWritable. Zmusza to użytkownika do każdorazowego parsowania danych do postaci klasy $FourInts$ lub $MultipleFourInts$. Dodatkowo użytkownik sam musi zaimplementować wymienione klasy, a serializacja obiektów do tekstu nie jest najszybszym możliwym sposobem [3]. Zarządzanie maszynami, precyzyjne wysyłanie danych oraz przesył różnych typów obiektów również nie należą do najłatwiejszych. Fakt potrzeby stworzenia dodatkowej rundy w celu policzenia statystyk prefiksowych dla całych maszyn, znacząco zmniejsza wydajność algorytmu. Największą jednak wadą powyższej implementacji jest brak generyczności. Użytkownik chcąc zmodyfikować algorytm lub jego dane wejściowe, musi zmienić znaczną część implementacji.

4.2.2. Implementacja algorytmu z wykorzystaniem biblioteki

Jednym z problemów napotkanych przy wcześniejszej implementacji algorytmu jest brak efektywnych serializatorów danych. W rozwiązaniu używającym biblioteki skorzystałem z serializatora Avro. Schemat obiektów wejściowych został przedstawiony na Listingu 4.10. Dzięki bibliotece *avro − tools* klasa *FoutsInts* - Listing 4.11 została wygenerowana automatycznie na podstawie schematu. Więcej na temat formatu danych i serializatorów w podrozdziale 4.2.3.1.

Listing 4.10: Schemat AVRO obiektów wejściowych

```
[{"namespace": "minimal_algorithms.examples.types",
 "type": "record",
 "name": "FourInts",
 "fields": [
    {"name": "first", "type": "int"},
    {"name": "second", "type": "int"},
    {"name": "third", "type": "int"},
    {"name": "fourth", "type": "int"}
  ]
}]
```

Listing 4.11: Wygenerowana klasa FourInts

```
@org.apache.avro.specific.AvroGenerated
public class FourInts
    extends org.apache.avro.specific.SpecificRecordBase
    implements org.apache.avro.specific.SpecificRecord {

  public static final org.apache.avro.Schema SCHEMA$ = ... ;
  public static org.apache.avro.Schema getClassSchema() {return SCHEMA$;}
  public int first;
  public int second;
  public int third;
  public int fourth;

  ... reszta wygenerowanego kodu ...
```

Listing 4.12: Komparator obiektów wejściowych

```
public class FourIntsCmp implements Comparator<FourInts> {
  @Override
  public int compare(FourInts o1, FourInts o2) {
    return o1.first > o2.first ? 1 : (o1.first < o2.first ? -1 : 0);
  }
}
```

Biblioteka do implementacji algorytmów minimalnych oferuje także wsparcie do obliczania łącznych statystyk. Podobnie jak na Listingu 4.10 użytkownik tworzy schemat statystyk (Listing 4.13), a potem automatycznie generuje odpowiadającą mu klasę (Listing 4.14). Następnie należy zmienić nadklasę na *StatisticsAggregator* oraz zaimplementować dwie metody *init* i *merge*. Dzięki klasie *SumSAFourInts* użytkownik nie musi implementować klasy *IndexedStatistics* z Listingu 4.3 oraz przejmować się sposobem przesyłania statystyk. Definiuje jedynie sposób ich inicjowania oraz łączenia, a biblioteka zajmuje się resztą oraz udostępnia przydatne metody operowania na nich. Więcej informacji na temat obsługi statystyk w bibliotece znajduje się w podrozdziale 4.2.3.4.

Listing 4.13: Schemat AVRO statystyk na obiektach wejściowych

```
[{"namespace": "minimal_algorithms.examples.types",
 "type": "record",
 "name": "SumSAFourInts",
 "fields": [
    {"name": "sum", "type": "int"}
  ]
}]
```

Listing 4.14: Klasa statystyk na obiektach FourInts

```
public class SumSAFourInts extends StatisticsAggregator {
  ... kod wygenerowany automatycznie z wykorzystaniem avro-tools ...

  public void init(GenericRecord record) {
    FourInts fourInts = (FourInts) record;
    this.sum = fourInts.getSecond() + fourInts.getThird() +
               fourInts.getFourth();
  }

  public StatisticsAggregator merge(StatisticsAggregator that) {
    if (that instanceof SumSAFourInts) {
      return new SumSAFourInts(this.sum + ((SumSAFourInts) that).getSum());
    }
    throw new org.apache.avro.AvroRuntimeException(
      "Trying to merge" + that.getClass().getName() + " with SumSAFourInts");
  }
}
```

Na Listingach 4.15 i 4.16 przedstawiono sposób uruchamiania programu. Biblioteka do implementacji algorytmów minimalnych udostępnia klasy konfiguracji (linia 10 Listingu 4.15 oraz linia 11 Listingu 4.16), dzięki którym użytkownik nie musi własnoręcznie wyliczać parametrów algorytmów oraz jest pewny, że przekazał wszystkie niezbędne zmienne.

Na Listingu 4.16 linia 20 pokazano jak skorzystać z gotowych algorytmów minimalnych wchodzących w skład biblioteki. Zauważmy, że łączenie rund różnych algorytmów minimalnych zaimplementowanych przy użyciu biblioteki jest zwięzłe i intuicyjne. Metoda *teraSort* umożliwia sortowanie dowolnych obiektów oraz oszczędza czas i wysiłek włożony w napisanie własnego algorytmu sortowania, tak jak miało to miejsce w implementacji bez użycia algorytmu. Dalsze informacje, na temat ułatwień wynikających z biblioteki, znajdują się w podrozdziale 4.2.3.5.

Listing 4.15: Uruchamianie algorytmu - część 1

```
public class PrefixApp extends Configured implements Tool {
  public int run(String[] args) throws Exception {
    if (args.length != 6) {
      System.err.println("Usage: PrefixApp" +
        "<home_dir> <input_path> <output_path> <items_no> " +
        "<partitions_no> <reduce_tasks_no>");
      return -1;
    }

    IOConfig ioConfig = new IOConfig(new Path(args[0]), new Path(args[1]),
      new Path(args[2]), FourInts.getClassSchema());
    Config config = new Config(getConf(), Integer.parseInt(args[3]),
      Integer.parseInt(args[4]), Integer.parseInt(args[5]));
    Schema statsSchema = SumSAFourInts.getClassSchema();
    return new HadoopMinAlgFactory(config).
      prefix(ioConfig, new FourIntsCmp(), statsSchema);
  }

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

Listing 4.16: Uruchamianie algorytmu - część 2

```
public class HadoopMinAlgFactory {
  private final String SORTING_DIR = "/sorting_output";

  public int teraSort(Path homeDir, Path input,
    Path output, BaseConfig baseConfig) throws Exception {
    ...
  }

  public int prefix(IOConfig ioConfig, Comparator cmp, Schema statsAggSchema)
      throws Exception {
    StatisticsConfig statisticsConfig = new StatisticsConfig(
      config, cmp, ioConfig.getBaseSchema(), statsAggSchema);
    return prefix(ioConfig.getHomeDir(), ioConfig.getInput(),
      ioConfig.getOutput(), statisticsConfig);
  }

  public int prefix(Path homeDir, Path input, Path output,
      StatisticsConfig statisticsConfig) throws Exception {
    Path sortingDir = new Path(homeDir + "/tmp" + SORTING_DIR);
    int ret = teraSort(homeDir, input, sortingDir, statisticsConfig);
    ret = ret == 0 ?
      PhasePrefix.run(sortingDir, output, statisticsConfig) : ret;
    Utils.deleteDirFromHDFS(conf, sortingDir, true);
    return ret;
  }
}
```

Listing 4.17 przedstawia rundę obliczania statystyk prefiksowych. W odróżnieniu od Listingu 4.8 oraz Listingu 4.9 runda jest zaimplementowana w pojedynczym kroku MapReduce. Zawdzięczamy to klasie *SendWrapper*, która umożliwia równoległe przesyłanie różnych typów obiektów między fazami *map* i *reduce*. W linii 10 ustawiamy schematy obiektów, które chcemy przesyłać równolegle, a następnie w fazie *map* w liniach 37 oraz 41 wysyłamy je na

wybrane maszyny. W fazie *reduce*, linie 70 – 71, dzielimy odebrane obiekty na dwie grupy względem typów. Warto zauważyć, że w liniach 72 – 82 używamy obiektów z zachowaniem ich oryginalnych typów. Dodatkowo omijamy krok rozsyłania wszystkich obiektów na pamięć lokalną maszyn, tak jak miało to miejsce na Listingu 4.8 przy użyciu metody *addCacheFile*. Więcej informacji znajdziemy w podrozdziale 4.2.3.3.

Zauważmy, jak łatwe jest rozsyłanie obiektów na wybrane maszyny – linie 37 oraz 41. Idea przesyłu obiektów jest taka sama jak w implementacji algorytmu bez użycia biblioteki, jednak została opakowana w przejrzyste metody. Dokładny opis znajduje się w podrozdziale 4.2.3.2.

Implementacja pokazuje również siłę klasy *StatisticsUtils*. W linii 61 deklarujemy obiekt operujący na statystykach, a następnie w liniach 73, 79 oraz 81 w zwięzły sposób obliczamy statystyki. Warto podkreślić fakt, że wszystkie operacje są generyczne i zależą tylko i wyłącznie od schematów obiektów wejściowych i klasy statystyk zdefiniowanej przez użytkownika przy uruchomieniu programu.

Ważną częścią Listingu 4.17 jest ustawianie schematów. Odpowiada za to funkcja zdefiniowana w linii 2 – *setSchemas*, która przyjmuje jako argument konfigurację przekazaną przez użytkownika przy uruchomieniu algorytmu (Listingi 4.15 i 4.16). Dzięki ustawianiu schematów informujemy rundę algorytmu jakich obiektów generycznych powinna używać. Schematy muszą zostać ustawione w fazie *map* – linia 24, fazie *reduce* – linia 58 oraz przy uruchomieniu rundy $PhasePrefix$ – linia 90.

Listing 4.17: Faza liczenia statystyk prefiksowych

```
1  public class PhasePrefix {
2    private static void setSchemas(Configuration conf) {
3      Schema baseSchema = Utils.retrieveSchemaFromConf(
4        conf, StatisticsConfig.BASE_SCHEMA_KEY);
5      Schema statisticsAggregatorSchema = Utils.retrieveSchemaFromConf(
6        conf, StatisticsConfig.STATISTICS_AGGREGATOR_SCHEMA_KEY);
7      StatisticsRecord.setSchema(statisticsAggregatorSchema, baseSchema);
8      MultipleRecords.setSchema(baseSchema);
9      MultipleStatisticRecords.setSchema(StatisticsRecord.getClassSchema());
10     SendWrapper.setSchema(baseSchema, statisticsAggregatorSchema);
11   }
12
13   public static class PrefixMapper extends
14         Mapper<AvroKey<Integer>, AvroValue<MultipleRecords>,
15                AvroKey<Integer>, AvroValue<SendWrapper>> {
16
17     private Configuration conf;
18     private AvroSender sender;
19     private StatisticsUtils statsUtiler;
20
21     @Override
22     public void setup(Context ctx) {
23       conf = ctx.getConfiguration();
24       setSchemas(conf);
25       sender = new AvroSender(ctx);
26       statsUtiler = new StatisticsUtils(Utils.retrieveSchemaFromConf(
27         conf, StatisticsConfig.STATISTICS_AGGREGATOR_SCHEMA_KEY));
28     }
29
30     @Override
31     protected void map(AvroKey<Integer> key,
32         AvroValue<MultipleRecords> value, Context context)
33         throws IOException, InterruptedException {
```

```
34
35      StatisticsAggregator partStats =
36        statsUtiler.foldLeftRecords(value.datum().getRecords(), null);
37      sender.sendToAllHigherMachines(new SendWrapper(null, partStats),
38        key.datum());
39
40      for (GenericRecord record : value.datum().getRecords()) {
41        sender.send(key, new SendWrapper(record, null));
42      }
43    }
44  }
45
46  public static class PrefixReducer extends
47        Reducer<AvroKey<Integer>, AvroValue<SendWrapper>,
48                AvroKey<Integer>, AvroValue<MultipleStatisticRecords>> {
49
50    private AvroSender sender;
51    private Configuration conf;
52    private StatisticsUtils statsUtiler;
53    private Comparator<GenericRecord> cmp;
54
55    @Override
56    public void setup(Context ctx) {
57      this.conf = ctx.getConfiguration();
58      setSchemas(conf);
59      sender = new AvroSender(ctx);
60      cmp = Utils.retrieveComparatorFromConf(conf);
61      statsUtiler = new StatisticsUtils(Utils.retrieveSchemaFromConf(conf,
62        StatisticsConfig.STATISTICS_AGGREGATOR_SCHEMA_KEY));
63    }
64
65    @Override
66    protected void reduce(AvroKey<Integer> key,
67        Iterable<AvroValue<SendWrapper>> values, Context context)
68        throws IOException, InterruptedException {
69
70      Map<Integer, List<GenericRecord>> groupedRecords =
71        SendingUtils.partitionRecords(values);
72      StatisticsAggregator priorPartitionStatistics =
73        statsUtiler.foldLeftAggregators(groupedRecords.get(2));
74
75      List<GenericRecord> elements = groupedRecords.get(1);
76      if (elements != null) {
77        java.util.Collections.sort(elements, cmp);
78        List<StatisticsAggregator> statistics = statsUtiler.
79          scanLeftRecords(elements, priorPartitionStatistics);
80        List<StatisticsRecord> statsRecords =
81          statsUtiler.zip(statistics, elements);
82        sender.send(key, new MultipleStatisticRecords(statsRecords));
83      }
84    }
85  }
86
87  public static int run(Path input, Path output,
88                        StatisticsConfig statsConfig) throws Exception {
89    Configuration conf = statsConfig.getConf();
90    setSchemas(conf);
91
92    Job job = Job.getInstance(conf, "JOB: Phase Prefix");
93    job.setJarByClass(PhasePrefix.class);
```

```
94      job.setNumReduceTasks(Utils.getReduceTasksCount(conf));
95      job.setMapperClass(PrefixMapper.class);
96
97      FileInputFormat.setInputPaths(job, input + "/" +
98        StatisticsConfig.SORTED_DATA_PATTERN);
99      FileOutputFormat.setOutputPath(job, output);
100
101     job.setInputFormatClass(AvroKeyValueInputFormat.class);
102     AvroJob.setInputKeySchema(job, Schema.create(Schema.Type.INT));
103     AvroJob.setInputValueSchema(job, MultipleRecords.getClassSchema());
104
105     job.setMapOutputKeyClass(AvroKey.class);
106     job.setMapOutputValueClass(AvroValue.class);
107     AvroJob.setMapOutputKeySchema(job, Schema.create(Schema.Type.INT));
108     AvroJob.setMapOutputValueSchema(job, SendWrapper.getClassSchema());
109
110     job.setReducerClass(PrefixReducer.class);
111     job.setOutputFormatClass(AvroKeyValueOutputFormat.class);
112     job.setOutputKeyClass(AvroKey.class);
113     job.setOutputValueClass(AvroValue.class);
114     AvroJob.setOutputKeySchema(job, Schema.create(Schema.Type.INT));
115     AvroJob.setOutputValueSchema(job,
116       MultipleStatisticRecords.getClassSchema());
117
118     return job.waitForCompletion(true) ? 0 : 1;
119   }
120 }
```

Porównując implementacje bez użycia biblioteki oraz z jej wykorzystaniem, możemy stwierdzić, że ta druga jest krótsza, efektywniejsza i posiada większą siłę wyrazu. Efektywność wynika z faktu zmniejszenia liczby rund MapReduce o jeden, pominięcia zapisu do plików oraz wykorzystania serializatora Avro. Przez siłę wyrazu mam na myśli generyczność implementacji. Użytkownik może łatwo modyfikować dane wejściowe, klasy obliczające statystyki oraz komparatory. Dodatkowo może wykorzystywać wcześniej zaimplementowane rundy MapReduce, np. *TeraSort*.

4.2.3. Opis implementacji biblioteki

Biblioteka na platformę Hadoop została napisana w języku Java 8. Do zarządzania projektem został użyty Apache Maven, program automatyzujący budowę oprogramowania. Odnośnie modułów Hadoopa, rozwiązanie jest oparte o Hadoop 2.8, wspieranego rozproszonym systemem plików HDFS oraz YARN-em.

W podrozdziale 4.2.1 zaobserwowaliśmy trudności związane z implementacją algorytmów rozproszonych na Hadoopa. Należą do nich między innymi:

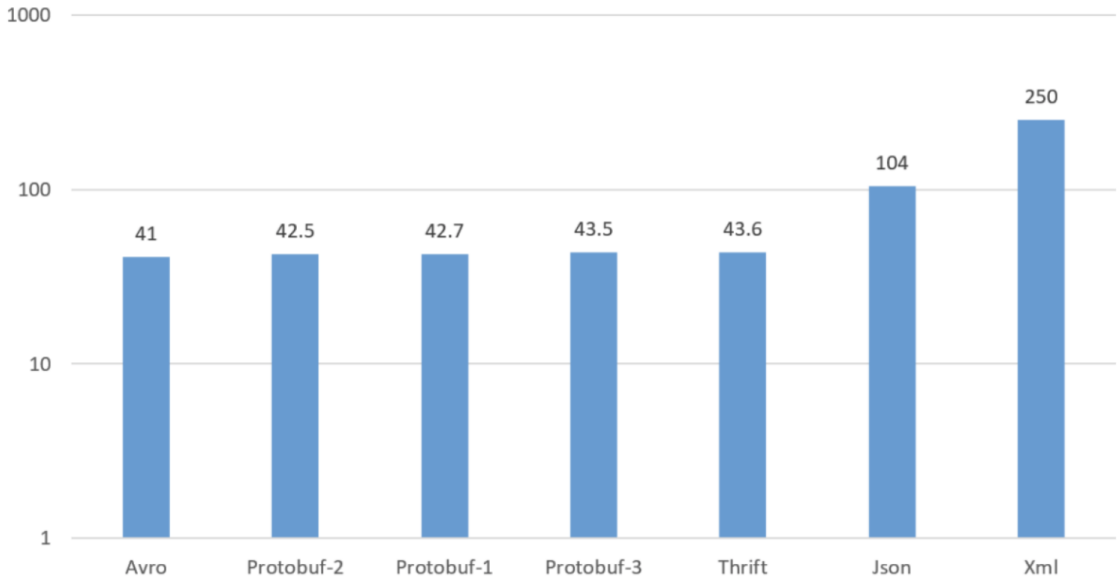
- brak efektywnej serializacji danych,
- brak intuicyjnego systemu zarządzania maszynami,
- brak precyzyjnego rozsyłania danych,
- brak efektywnego sposobu na równoległe przesyłanie różnych typów danych między fazami MapReduce,
- brak generyczności algorytmu.

W poniższych podrozdziałach przedstawię w jaki sposób biblioteka do implementacji algorytmów minimalnych rozwiązuje powyższe problemy.

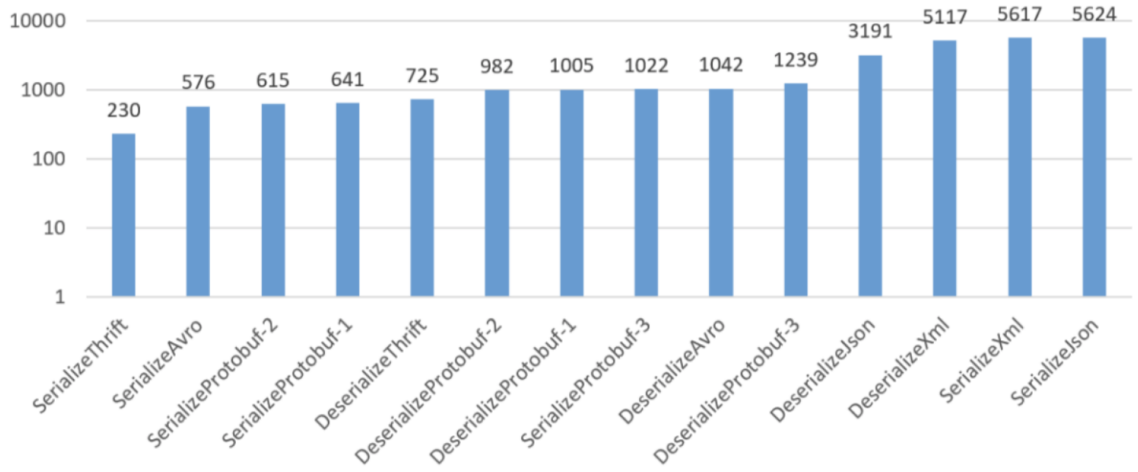
4.2.3.1. Format danych

Problemem, który napotykamy pisząc algorytmy na platformę Hadoop jest ilość zapisu i odczytu danych z dysku lokalnego maszyny. Podstawowa wersja Hadoopa zapisuje dane jako tekst. Jest to rozwiązanie wolne i zajmujące dużo pamięci [8, 12]. Dodatkowo przesyłanie skomplikowanych obiektów jako tekst jest uciążliwe. Zazwyczaj wymaga implementacji własnych klas serializujących. Alternatywą są gotowe biblioteki serializatorów wspierające rozpoznawanie zapisywanych typów danych. Są to biblioteki odpowiedzialne za tłumaczenie struktur danych i obiektów do postaci umożliwiającej zapis, odczyt i przesył, tak aby mogłby zostać odtworzone do identycznego stanu jak początkowy. Dodatkowo przy zapisie obiektu, uwzględniany jest jego typ, zatem liczba 123456 zostanie zserializowana do 4 bajtów, a nie 6 bajtów tak jakby miało to miejsce w przypadku tekstu. Istnieje cała gama formatów serializacji danych, między innymi: Protobuf, Thrift i Avro. Na Rysunkach 4.1 oraz 4.2 przedstawiono porównanie serializatorów. Testy zostały przeprowadzone na dużych obiektach, typowych dla algorytmów MapReduce [1, 3].

Rysunek 4.1: Wielkości plików po serializacji w MB



Rysunek 4.2: Czasy serializacji i deserializacji w milisekundach



Z przedstawionych wykresów widać, że Avro, Protobuf i Thrift należą do czołówki serializatorów. W moim rozwiązaniu użyłem Avro, którego podstawowymi zaletami są [1, 2, 5]:

- szybkość zapisu i odczytu,
- kompresja danych do formatu binarnego,
- możliwość serializowania skomplikowanych obiektów,
- elastyczność schematów opisujących struktury serializowanych danych – schematy zapisu i odczytu mogą być różne,
- klasa abstrakcyjna *GenericData.Record* umożliwiająca generyczne operowanie na obiektach typu Avro.

Najważniejszy jest jednak fakt, że dzięki pakietowi Avro oraz możliwości dynamicznego tworzenia klas na podstawie schematów, biblioteka algorytmów minimalnych na Hadoopie jest generyczna. Każda funkcja i faza MapReduce może zostać zaaplikowana do dowolnego typu danych. Jest to ogromna zaleta biblioteki, ponieważ typowe programy MapReduce są pisane pod konkretne typy danych, a zmiana danych pociąga za sobą konieczność poprawiania programu.

4.2.3.2. Zarządzanie maszynami

W przedstawionych algorytmach minimalnych można zauważyć jak ważna jest numeracja maszyn wchodzących w skład klastra. Niezwykle istotne jest utrzymywanie kolejności obiektów między maszynami. Dodatkowo fazy MapReduce wymagają wysyłania obiektów na konkretne maszyny. Także niektóre obliczenia wykonywane są tylko na wybranych maszynach.

Niestety Hadoop nie umożliwia ponumerowania lub oznaczania maszyn. Wszystkie maszyny oprócz *NameNode* są równoważne i to YARN odpowiada za docelowe rozmieszczenie plików. Zaletą takiego rozwiązania jest fakt, że YARN wybierze maszyny, które są najbliżej danych i przesył informacji zostanie zoptymalizowany.

Rozwiązaniem, które zastosowałem jest wykorzystanie klucza, z pary wysyłanej przez MapReduce, jako numeru maszyny. Wszystkie fazy MapReduce zaimplementowane w bibliotece operują na parach *<numerMaszyny, obiekty>*. Dzięki takiemu rozwiązaniu wszystkie

obiekty, które powinny znaleźć się na maszynie M_i, zostaną przetworzone przez jeden proces *reduce*.

Niech $i, j \in [0,\ liczbaMaszyn]$. W ramach biblioteki zostały zaimplementowane metody umożliwiające wysyłanie obiektów na:

- konkretną maszynę i,
- wszystkie maszyny w przedziale $[i, j)$,
- wszystkie maszyny większe niż i,
- wszystkie maszyny mniejsze niż i,
- wszystkie maszyny w systemie.

Funkcje zostały napisane w oparciu o klasy *Mapper.Context* oraz *Reducer.Context*, które są standardowym sposobem przesyłu niewielkich danych w Hadoopie.

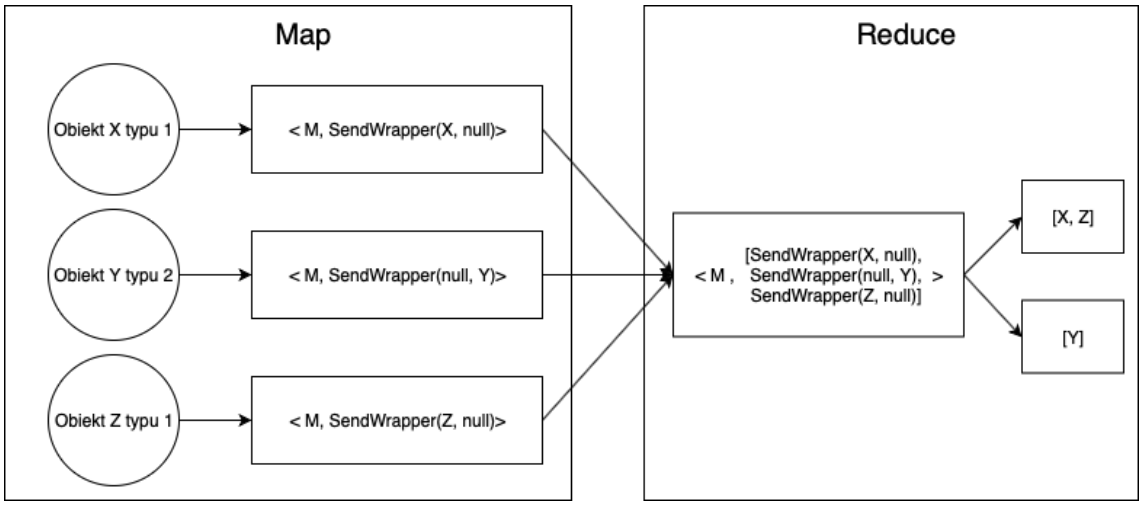
4.2.3.3. Przesył różnych typów obiektów

Implementując niektóre algorytmy minimalne napotykamy problem jednoczesnego przesyłania różnych typów obiektów między fazami lub rundami MapReduce. Standardowo funkcje *map* i *reduce* pozwalają na przesył jednego typu elementów. Nie istnieje także klasa krotek działająca na obiektach generycznych.

W tym miejscu z pomocą przychodzi serializator Avro. Biblioteka algorytmów minimalnych implementuje klasę *SendWrapper*, pozwalającą na wysyłanie zmieszanych typów danych. Na ten moment wspierana jest obsługa tylko dwóch typów obiektów. Rozszerzenie klasy na więcej typów jest jednak wykonalne i byłoby kompatybilne z obecną wersją biblioteki.

Klasa *SendWrapper* opakowuje dwa obiekty Avro w jeden, tworząc schemat danych dynamicznie. Utworzony schemat posiada dwa opcjonalne pola. Gdy opakowujemy obiekt pierwszego typu, to drugie pole jest wartością *null* i odwrotnie dla opakowywania elementu drugiego typu. Przesyłanie wartości *null* jest niewielkim kosztem, który ponosimy za wspieranie funkcjonalności wysyłania dwóch typów obiektów. Dodatkowo biblioteka udostępnia funkcje pozwalające na filtrowanie elementów pierwszego / drugiego typu. Rysunek 4.3 przedstawia ogólną zasadę działania klasy $SendWrapper$, a Listing 4.17 konkretny sposób zastosowania.

Rysunek 4.3: Przesyłanie dwóch typów obiektów



Alternatywą dla tego rozwiązania jest zapisywanie obiektów jednego z typów do plików na lokalnej pamięci maszyny, a następnie odczytywanie ich w fazie *reduce*. Wadą takiego sposobu jest fakt, że rezygnujemy z automatycznego obsługiwania plików przez MapReduce oraz wykonujemy więcej operacji odczytu i zapisu.

Innym rozwiązaniem jest stworzenie dodatkowej rundy MapReduce, w której zapiszemy przetworzone obiekty do HDFS, a następnie odczytamy je w odpowiedniej fazie innej rundy algorytmu. Jest to rozwiązanie wolne i wymagające dodatkowej pracy.

4.2.3.4. Statystyki

Kolejnym zagadnieniem jest implementacja funkcji statystyk łącznej na zdefiniowanym zbiorze obiektów. Rozwiązaniem, które zastosowałem jest stworzenie abstrakcyjnej klasy *StatisticsAggregator* deklarującej dwie funkcje abstrakcyjne:

1. *public abstract void init(GenericRecord record)* - definiuje tworzenie statystyk na podstawie obiektu. Klasa *GenericRecord* jest klasą abstrakcyją obiektów Avro.

2. *public abstract StatisticsAggregator merge(StatisticsAggregator that)* - opisuje sposób łączenia dwóch statystyk, $stat(S) = merge(stat(S_1), stat(S_2))$, gdzie S_1 i S_2 tworzą podział S, czyli $S_1 \cup S_2 = S$ oraz $S_1 \cap S_2 = \emptyset$.

Użytkownik powinien stworzyć podklasę klasy *StatisticsAggregator* i napisać definicje powyższych funkcji abstrakcyjnych. Proces implementacji składa się z następujących kroków i został przedstawiony na Listingach 4.13 oraz 4.14:

1. Stworzenie schematu *.avsc* dla klasy statystyk. Obiekty klasy statystyk są regularnymi obiektami, które są serializowalne i przesyłane między fazami MapReduce.

2. Wygenerowanie klasy w Javie na podstawie pliku *.avsc*.

3. Zmodyfikowanie klasy utworzonej w kroku 2. Należy:

   - podziedziczyć po klasie *StatisticsAggregator*,
   - zaimplementować funkcje *init* oraz *merge*.

   Zaletami takiego podejścia są:

- definiowanie dowolnie skomplikowanych funkcji statystyk,

- obliczanie jednocześnie kilku funkcji statystyk przy jednej implementacji klasy reprezentującej statystyki,

- przechowywanie dodatkowych informacji związanych ze statystykami.

  Biblioteka udostępnia także funkcje agregujące statystyki takie jak:

- *scanLeft* – statystyki prefiksowe listy obiektów,

- *foldLeft* – agregacja statystyk z listy do pojedynczej wartości.

Dodatkowo została stworzona klasa *RangeTree* umożliwiająca liczenie statystyk na przedziałach, w sposób sekwencyjny na pojedynczej maszynie. *RangeTree* jest w pełni serializowalne co pozwala na przesyłanie uszeregowanych statystyk, a następnie wykonywanie optymalnych zapytań. Klasa implementuje klasyczne, pełne drzewo binarne oraz funkcje:

- *public void insert(StatisticsAggregator element, int pos)* – wrzucanie statystyk do drzewa,
- *public StatisticsAggregator query(int start, int end)* – agregacja statystyk z zakresu $[start, end)$.

4.2.3.5. Użytkowanie

Pisanie programów na platformę Hadoop wymaga wiedzy i dokładności. Standardowo na początku użytkownik ustawia w konfiguracji programu wszelkie niezbędne parametry. Jednak system nie sprawdzana ich obecności i poprawności. Brak jednego z nich skutkuje zatrzymaniem wykonania programu i czasem straconym na znalezienie brakujących zmiennych.

W bibliotece algorytmów minimalnych wymagamy od użytkownika uzupełnienia wszystkich niezbędnych parametrów, a brak któregoś z nich jest sygnalizowany czytelnym komunikatem. Dodatkowo sprawdzana jest ich poprawność, aby uchronić użytkownika od banalnych błędów.

Biblioteka udostępnia także API do zarządzania plikami Avro na pamięci lokalnej maszyny. W jego skład wchodzą funkcje takie jak: *zapis*, *odczyt*, *usuwanie*, ale także *łączenie* dwóch plików Avro.

Dodatkowo istnieje funkcja, operująca na konfiguracji programu, zapisująca i odczytująca komparator obiektów Avro. Dzięki tej metodzie użytkownik raz zapisuje komparator do konfiguracji, a następnie może z niego korzystać w dowolnej fazie MapReduce.

4.3. Spark

4.3.1. Implementacja algorytmu bez użycia biblioteki

Na Listingu 4.20 przedstawiono implementację algorytmu statystyk prefiksowych bez użycia biblioteki. Dane wejściowe należały do klasy ukazanej na Listingu 4.18. Funkcję porównującą obiekty klasy $FourInts$ przedstawiono na Listingu 4.19. Funkcją statystyk była suma wartości zwracanych przez funkcję $getValue()$ – Listing 4.18, linia 6.

Listing 4.18: Klasa danych wejściowych do algorytmu statystyk prefiksowych

```
1 class FourInts(first: Int, second: Int, third: Int, fourth: Int) extends
      Serializable {
2   def getFirst(): Int = this.first
3   def getSecond(): Int = this.second
4   def getThird(): Int = this.third
5   def getFourth(): Int = this.fourth
6   def getValue(): Int = this.getSecond() + this.getThird() + this.getFourth()
7 }
```

Listing 4.19: Funkcja porównująca obiekty klasy FourInts

```
1 object FourInts {
2   def cmpKey(o: FourInts): Int = o.getFirst()
3 }
```

Omówmy zwięźle Listing 4.20. W liniach 6 – 7 zaimplementowano algorytm $TeraSort$, czyli rundy 1 – 2 algorytmu statystyk prefiksowych. W tym celu użyto wbudowanej funkcji sortującej *sortBy*. Następnie w liniach 9 – 11 wykonujemy fazę *map-shuffle*, polegającą na obliczeniu i rozesłaniu statystyk prefiksowych dla całych partycji. Funkcja *partitionStatistics* iteruje po elementach RDD i wylicza sumę obiektów dla każdej partycji. Następnie w linii 10 obliczamy sumy prefiksowe całych partycji, a w linii 11 rozsyłamy je na wszystkie maszyny. W tym celu użyto wbudowanej funkcji *broadcast*, która wysyła argument na każdą maszynę. Jest to wygodny sposób udostępniania obiektów innym partycjom, potrzebnych na wielu etapach obliczeń. Niestety postępując według algorytmu, powinniśmy rozsyłać statystyki prefiksowe partycji tylko do maszyn o większych indeksach. Zaimplementowana wersja algorytmu nie traci swojej minimalności, jednak jest wolniejsza i zużywa więcej pamięci. W liniach 13 – 24 wykonujemy fazę *reduce*. Dzięki prostej funkcji statystyk oraz wcześniej udostępnionej liście *distPartStats* w łatwy sposób obliczamy końcowy wynik algorytmu.

Listing 4.20: Implementacja algorytmu statystyk prefiksowych bez użycia biblioteki

```
1  object PrefixApp {
2
3    def prefix(sc: SparkContext, rdd: RDD[FourInts], numPartitions: Int):
4          RDD[(Int, FourInts)] = {
5
6      val sortedRdd = rdd.repartition(numPartitions).
7          sortBy(FourInts.cmpKey).persist()
8
9      val partStats = partitionStatistics(sortedRdd).collect()
10     val prefixPartStats = partStats.scanLeft(0){(acc, o) => acc + o}
11     val distPartStats = sc.broadcast(prefixPartStats).value
12
13     sortedRdd.mapPartitionsWithIndex((pIndex, partitionIt) => {
14       if (partitionIt.hasNext) {
15         val elements = partitionIt.toList
16         val prefixes = elements.scanLeft(distPartStats(pIndex)){
17             (res, o) => res + o.getValue()
18         }
19         prefixes.zip(elements).iterator
20       } else {
21         Iterator()
22       }
23     })
24   }
25
26   def partitionStatistics(rdd: RDD[FourInts]): RDD[Int] = {
27     rdd.mapPartitions(partitionIt => {
28       if (partitionIt.isEmpty) {
29         Iterator()
30       } else {
31         Iterator(partitionIt.foldLeft(0){(acc, o) => acc + o.getValue()})
32       }
33     })
34   }
35 }
```

4.3.2. Implementacja algorytmu z wykorzystaniem biblioteki

Pisanie programów w Sparku jest łatwe i intuicyjne. Takie założenia spełnia też biblioteka do implementacji algorytmów minimalnych. Punktem wejściowym jest podanie dwóch argumentów (Listing 4.22):

- *SparkSession* – obiekt niezbędny do uruchomienia programu w Sparku,
- *numberOfPartitions* – liczba maszyn (partycji).

W przypadku algorytmów wymagających tylko posortowania, takich jak: *TeraSort*, *ranking*, *pół-złączenia* użytkownik musi zdefiniować komparator obiektów wejściowych. Komparator jest pojedynczą funkcją, co pozwala na używanie wielu z nich przy raz wczytanych danych – linia 1, Listing 4.21.

Natomiast algorytmy agregujące statystyki, czyli: *statystyki prefiksowe*, *grupowanie*, *statystyki okienkowe* wymuszają dodatkowe zaimplementowanie podklasy *StatisticsAggregator* oraz zdefiniowanie funkcji *init*, tak jak zostało to pokazane na Listingu 4.21, odpowiednio linie 4 – 12 oraz 2. Więcej informacji na temat obliczania statystyk w Sparku znajduje się w podrozdziale 4.3.3.3.

Listing 4.21: Funkcja porównująca oraz funkcja statystyk

```
1  val cmpKey = (o: FourInts) => o.getFirst
2  val sumAgg = (o: FourInts) => new SumAggregator(o.getValue)
3
4  class SumAggregator(value: Int)
5      extends StatisticsAggregator[SumAggregator] {
6
7    override def merge(that: SumAggregator): SumAggregator = {
8      new SumAggregator(this.value + that.getValue)
9    }
10
11   def getValue: Int = value
12 }
```

Omówmy precyzyjniej Listing 4.22. W liniach 4 – 7 widzimy generyczną deklarację funkcji $prefix$. Dzięki takiej formie, użytkownik może aplikować do funkcji dowolnie skomplikowane obiekty, które są porównywalne i posiadają funkcję zwracającą statystyki pojedynczego obiektu. W linii 9 sortujemy RDD za pomocą algorytmu $TeraSort$, wykorzystującego wbudowaną w Sparka funkcję $sortBy$.

Następnie w liniach 11 – 12, dzięki wbudowanej w bibliotekę funkcji $partitionStatistics$, obliczamy statystyki dla całych partycji. Wynik tych obliczeń łączymy z indeksami maszyn, które będą dolnymi ograniczeniami w przesyłaniu statystyk. Tak przygotowaną listę rozsyłamy do odpowiednich maszyn (linie 14 – 15). Funkcja $sendToAllHigherMachines$ jako argumenty przyjmuje obiekt $SparkSession$, listę obiektów z dolnym ograniczeniem przesyłu oraz ilość wszystkich maszyn, czyli górne ograniczenie przesyłu. RDD $distPartitionStatistics$ zawiera rozesłane statystki całych partycji zgodnie z opisem w podrozdziale 3.4, runda 3, faza $map - shuffle$. Dokładne wyjaśnienie implementacji rozsyłania danych znajduje się w podrozdziale 4.3.3.1. Warto zauważyć, że pozbyliśmy się funkcji $broadcast$ wykorzystanej na Listingu 4.20, tym samym zoptymalizowaliśmy implementację algorytmu.

W ostatnim kroku, linie 17 – 32, wykorzystujemy metodę $zipPartitions$ w celu równoległego iterowania po posortowanych obiektach wejściowych oraz wcześniej rozesłanych statystykach całych partycji. Usunięcie funkcji $broadcast$ zmusiło nas do iteracji po dwóch RDD zawierających różne typy danych. Jednak odpowiednia implementacja rozsyłania obiektów oraz wbudowana w Sparka metoda $zipPartitions$ umożliwiają nam łatwe i efektywne implementowanie tej funkcjonalności. Dokładniejszy opis problemu i rozwiązania znajduje się w podrozdziale 4.3.3.2.

Do obliczenia wyniku dla każdego obiektu wykorzystujemy funkcje $scanLeft$ oraz $foldLeft$ z klasy $StatisticsUtils$ wchodzącej w skład biblioteki. Ich użycie jest bardzo intuicyjne i przydatne w liczeniu generycznych, łącznych statystyk. Więcej na temat statystyk w podrozdziale 4.3.3.3.

Listing 4.22: Implementacja algorytmu statystyk prefiksowych z wykorzystaniem biblioteki

```
class MinimalAlgorithm(spark: SparkSession, numberOfPartitions: Int) {
  protected val sc: SparkContext = spark.sparkContext

  def prefix[T, K, S <: StatisticsAggregator[S]]
        (rdd: RDD[T], cmpKey: T => K, statsAgg: T => S)
        (implicit ord: Ordering[K], ttag: ClassTag[T],
         ktag: ClassTag[K], stag: ClassTag[S]): RDD[(S, T)] = {

    val sortedRdd = teraSort(rdd, cmpKey).persist()

    val partStatsWithBounds = StatisticsUtils.partitionStatistics(
        sortedRdd, statsAgg).collect().zip(List.range(0, numPartitions))

    val distPartitionStatistics = Utils.sendToAllHigherMachines(
        sc, partStatsWithBounds, numPartitions)

    sortedRdd.zipPartitions(distPartitionStatistics) {
        (partitionIt, partitionStatisticsIt) => {
      if (partitionIt.hasNext) {
        val elements = partitionIt.toList
        val statistics = elements.map{e => statsAgg(e)}
        val prefixes = if (partitionStatisticsIt.hasNext) {
          val startValue = StatisticsUtils.foldLeft(partitionStatisticsIt)
          StatisticsUtils.scanLeft(statistics, startValue).drop(1)
        } else {
          StatisticsUtils.scanLeft(statistics)
        }
        prefixes.zip(elements).iterator
      } else {
        Iterator()
      }
    }}
  }
}
```

Na podstawie Listingu 4.22 możemy stwierdzić, że implementacja generycznego algorytmu statystyk prefiksowych jest zwięzła i intuicyjna. Ilość kodu jest porównywalna do implementacji z Listingu 4.20, jednak siła wyrazu Listingu 4.22 jest dużo większa. Użytkownik z łatwością może testować algorytm na różnych danych wejściowych, funkcjach statystyk czy komparatorach. Dodatkowo poprawiliśmy złożoność obliczeniową i pamięciową, rozsyłając statystyki prefiksowe maszyn tylko na wybrane partycje.

4.3.3. Opis implementacji biblioteki

Biblioteka jest oparta o wersję Spark 2.3 i jest napisana w języku Scala 2.11. Do zarządzania projektem użyto Apache Maven, programu automatyzującego budowę oprogramowania.

W podrozdziale 4.3.1 zaobserwowaliśmy trudności związane z implementacją minimalnych algorytmów rozproszonych na Sparka. Głównymi z nich są:

- brak precyzyjnego rozsyłania danych,
- słaba generyczności algorytmu.

W poniższych podrozdziałach przedstawię w jaki sposób biblioteka rozwiązuje powyższe problemy oraz pomaga implementować algorytmy minimalne.

4.3.3.1. Zarządzanie maszynami

W Sparku nie możemy operować bezpośrednio na maszynach. Przykrywa je bowiem warstwa abstrakcji w postaci kolekcji rozproszonych danych, czyli RDD. Podstawową jednostką RDD są partycje. W moim rozwiązaniu to właśnie partycje symulują maszyny przedstawiane w algorytmach minimalnych.

RDD w API Sparka jest reprezentowane jako kolekcja partycji. Z tego wynika, że kolejne partycje mogą odpowiadać kolejnym maszynom. W mojej bibliotece wysyłanie obiektów na wybrane maszyny polega na zapisaniu ich na docelowych partycjach. Nie możemy jednak przesyłać pojedynczych elementów. Wszystkie operacje na RDD muszą być wykonywane przy pomocy transformacji. Rozwiązanie, które zaimplementowałem opiera się na przetasowywaniu RDD, a następnie jego podziale na partycje za pomocą zdefiniowanych podklas klasy *Partitioner*. Biblioteka do implementacji algorytmów minimalnych zawiera dwie podklasy:

1. *KeyPartitioner* – podział RDD wyznacza klucz obiektu, który jest docelowym indeksem maszyny,
2. *PerfectPartitioner* – na podstawie rankingu obiektów dzieli RDD na partycje o równym rozmiarze, zgodnie z opisem w sekcji 3.7.

Powyższe rozwiązanie nie gwarantuje jednak zachowania kolejności elementów. RDD są stałe, czyli raz stworzone nie mogą ulec zmianie. Jednak nowe RDD są tworzone przez transformacje istniejących. Przekształcenia takie jak:

- *map* – aplikowanie funkcji do obiektów na RDD,
- *filter* – filtrowanie obiektów RDD,
- *mapPartitions* – aplikowanie funkcji do całych partycji.

utrzymują porządek na partycji. Natomiast istnieją też transformacje zaburzające kolejność elementów. Należą do nich między innymi:

- *sortBy* – sortowanie elementów,
- *partitionBy* – podział RDD na nowe partycje za pomocą zdefiniowanej klasy podziału.

W przypadku wysyłania obiektów na wszystkie maszyny, została zaimplementowana też druga, konkurencyjna metoda. Wykorzystuje ona metodę *broadcast*. Polega na natychmiastowym stworzeniu stałej, zapisanej w pamięci podręcznej każdego węzła klastra. Spark optymalizuje wysyłanie takich zmiennych, zatem jest to dobra opcja do przesłania kopii dużych zbiorów danych potrzebnych na wielu etapach obliczeń. W sytuacji, gdy kopie wszystkich elementów są potrzebne tylko w następnej transformacji, lepiej użyć leniwie wykonującego się *Partitionera* z pierwszego rozwiązania [4].

Tak jak w przypadku Hadoopa, udostępnione zostały następujące funkcje przesyłu obiektów na:

- konkretną maszynę i,
- wszystkie maszyny w przedziale $[i, j)$,
- wszystkie maszyny większe niż i,
- wszystkie maszyny mniejsze niż i,
- wszystkie maszyny w systemie,

gdzie $i, j \in [0,\ liczbaMaszyn]$.

4.3.3.2. Przesył różnych typów obiektów

Na platformie Spark również doświadczamy problemu przesyłania różnych typów obiektów między transformacjami RDD. W tej sekcji przedstawię możliwe rozwiązania tego problemu oraz uargumentuję wybór zaimplementowanej metody.

Pierwszym ze sposobów jest użycie metody *broadcast*. Jak już wspomniałem wcześniej, jest to dobre rozwiązanie, gdy chcemy używać przesyłanych obiektów na wszystkich maszynach i w dodatku w kilku transformacjach. W przeciwnym przypadku marnujemy tylko zasoby Sparka.

Inną możliwością jest wykorzystanie klasy *Accumulator* udostępnianej przez Sparka. Obiekty tej klasy są współdzielonymi akumulatorami, do których można dodawać obiekty tylko podczas przemiennej, asocjacyjnej transformacji np: *map*. Co więcej, węzły wykonawcze (*DataNode*) mogą tylko zapisywać dane, a jedynie węzeł główny (*NameNode*) może je odczytywać [4]. Niestety oba powyższe fakty znacząco ograniczają możliwości użycia akumulatorów.

Kolejnym rozwiązaniem jest zapisywanie obiektów do rozproszonego systemu danych np: HDFS. Jednak w Sparku byłoby to ogromne spowolnienie dla całego programu. Standardowo zapis danych do pamięci następuje tylko w momencie utrwalenia ostatecznych rezultatów lub gdy brakuje pamięci operacyjnej [14].

Następnym sposobem jest rozwiązanie podobne do zaimplementowanego w bibliotece algorytmów minimalnych na Hadoopa. Sprowadza się ono do mieszania obiektów różnych typów na jednym RDD. Rozwiązanie wykorzystywałoby API Sparka, dzięki czemu byłoby łatwe w obsłudze i bardziej efektywne niż zapis do rozproszonego systemu danych. Niestety w momencie transformacji wymagałoby rozdzielenia obiektów, co generowałoby dodatkowe koszty czasowe.

Ostatnim i wybranym rozwiązaniem jest łączenie RDD. API Sparka udostępnia metodę *zipPartitions* pozwalającą na równoległe iterowanie po od dwóch do czterech RDD, zachowując przy tym kolejność partycji i obiektów. RDD muszą składać się z takiej samej liczby partycji. W momencie równoległego iterowania, partycje zostaną połączone w takiej kolejności, w jakiej występują na danym RDD. Zaimplementowanie równoległego przesyłu różnych typów obiektów składa się z następujących kroków:

1. Wysyłamy obiekty typu **I** na wybrane maszyny → powstaje nowe $Rdd1$ – na Listingu 4.22 jest nim $sortedRDD$,

2. Wysyłamy obiekty typu **II** na wybrane maszyny → powstaje nowe $Rdd2$ – na Listingu 4.22 jest nim $distPartitionStatistics$,

3. Łączymy *Rdd1* z *Rdd2* i aplikujemy transformację:

$$Rdd1.zipPartitions(Rdd2)\{(partitions1Iter,\ partitions2Iter) => \{...\}\}$$

   Za pomocą iteratorów $partitions1Iter$ oraz $partitions2Iter$ mamy dostęp do obiektów z partycji odpowiednio Rdd1 i Rdd2. Przykład użycia przedstawiono na Listingu 4.22 linie 19 – 31.

4.3.3.3. Statystyki

Problem liczenia statystyk na Sparku rozwiązałem w sposób bardzo podobny do tego na Hadoopie. Została stworzona klasa abstrakcyjna *StatisticsAggregator* deklarująca funkcję abstrakcyjną

$$def\ merge(statisticsAggregator : S) : S$$

gdzie S jest podklasą klasy *StatisticsAggregator*. Funkcja *merge*, tak samo jak w Hadoopie, definiuje proces łączenia dwóch statystyk. Po stronie użytkownika pozostaje stworzenie klasy S oraz zdefiniowanie funkcji $init : T \to S$, gdzie T jest typem obiektów wejściowych. Na Listingu 4.21 znajduje się przykład implementacji takiej klasy statystyk (linie 4 – 12) oraz funkcji *init* (linia 2). Definiowanie funkcji *init* poza klasą S umożliwia większą elastyczność i generalizację w użytkowaniu i pisaniu algorytmów.

Biblioteka do implementacji algorytmów minimalnych udostępnia także następujące operacje na statystykach:

- *scanLeft* – statystyki prefiksowe dla kolekcji,
- *foldLeft* – agregacja statystyk z kolekcji do pojedynczej wartości,
- *scanLeftPartitions* – statystyki prefiksowe na partycjach zawartych w RDD,
- *partitionStatistics* – statystyki dla każdej partycji zawartej w RDD.

Identycznie jak w rozwiązaniu na Hadoopa, została zaimplementowana klasa *RangeTree*, umożliwiająca liczenie statystyk na przedziałach, w sposób sekwencyjny na pojedynczej maszynie. Klasa implementuje klasyczne, pełne drzewo binarne oraz funkcje:

- *insert(element: StatisticsAggregator, position: Int): Unit* – wrzucanie statystyk do drzewa,
- *query(start: Int, end: Int): StatisticsAggregator* – agregacja statystyk z zakresu $[start, end)$.

Rozdział 5

Testy

Biblioteka do implementacji algorytmów minimalnych zawiera zestaw testów poprawnościowych sprawdzających działanie metod API i przykładowych algorytmów minimalnych w spreparowanym środowisku oraz skrypty testujące ich efektywność w warunkach produkcyjnych. Dodatkowo w ramach pracy przeprowadziłem testy wydajnościowe na zaimplementowanych algorytmach. Przy użyciu generatora losowych danych typu Avro (https://github.com/confluentinc/avro-random-generator) stworzyłem dwie grupy danych ważące 1 GB (10 000 000 obiektów) i 5 GB (45 000 000 obiektów). Obiekty zostały stworzone przy pomocy schematu z Listingu 5.1:

Listing 5.1: Schemat obiektów użytych podczas testowania

```
{ "type": "record",
  "name": "Complex",
  "namespace": "complex.type",
  "fields":
    [
      { "name": "null_prim", "type": ["null", "int"] },
      { "name": "boolean_prim", "type": "boolean" },
      { "name": "int_prim", "type": {
          "type": "int",
          "arg.properties": {
            "range": {
              "min": -10,
              "max": 10
            }
          }
        }
      },
      { "name": "long_prim", "type": "long" },
      { "name": "float_prim", "type": "float" },
      { "name": "double_prim", "type": "double" },
      { "name": "string_prim", "type": "string" },
      { "name": "bytes_prim", "type": "bytes" },
      { "name": "middle", "type":
        { "type": "record",
          "name": "MiddleNested",
          "fields": [
            { "name": "middle_array",
              "type": {
                "type": "array",
                "items": "float"
              }
            },
```

```
            { "name": "inner",
              "type": {
                "type": "record",
                "name": "InnerNested",
                "fields": [
                  { "name": "inner_int",
                    "type": "int"
                  },
                  { "name": "inner_string",
                    "type": "string"
                  }
                ]
              }
            }
          ]
        }
      }
    ]
}
```

Testy zostały przeprowadzone na dwóch typach klastrów składających się z:

- 1 NameNode + 5 DataNode
- 1 NameNode + 10 DataNode

Klastry zostały utworzone na platformie *Amazon Web Services*. Każdy NameNode i DataNode posiadał 4-rdzeniowy procesor oraz 6 GB pamięci RAM. Do porównywania obiektów został wykorzystany komparator z Listingu 5.2:

Listing 5.2: Komparator obiektów użyty podczas testowania

```
public class ComplexComparator implements Comparator<Complex> {
    @Override
    public int compare(Complex o1, Complex o2) {
        int innerInt1 = o1.getMiddle().getInner().getInnerInt();
        int innerInt2 = o2.getMiddle().getInner().getInnerInt();
        return innerInt1 > innerInt2 ?
            1 : (innerInt1 < innerInt2 ?
                -1 : o1.getLongPrim() > o2.getLongPrim() ?
                    1 : o1.getLongPrim() < o2.getLongPrim() ?
                        -1 : 0));
    }
}
```

5.1. Hadoop

W poniższych sekcjach zostały przedstawione wyniki testów poszczególnych algorytmów minimalnych wykonanych na Hadoopie. Kolumny mają następujące znaczenie:

- Runda – runda algorytmu MapReduce opisana w sekcji 3,
- Czas (s) – całkowity czas wykonania algorytmu,
- Max zużycie pamięci (GB) – maksymalne zużycie pamięci na całym klastrze podczas wykonywania algorytmu,
- Max przesył danych (GB) – maksymalny przesył danych wewnątrz całego klastra podczas wykonywania algorytmu.

5.1.1. TeraSort

Tablica 5.1: Wykonanie algorytmu *TeraSort*

Wielkość danych (GB)	Liczba maszyn	Runda	Czas (s)	Max zużycie pamięci (GB)	Max przesył danych (GB)
1	5	Próbkowanie	43	4,785	0,003
		Sortowanie	104	12,062	1,187
5	5	Próbkowanie	83	20,252	0,005
		Sortowanie	273	34,677	5,099
1	10	Próbkowanie	41	4,643	0,003
		Sortowanie	70	12,303	1,133
5	10	Próbkowanie	55	19,612	0,012
		Sortowanie	139	40,187	5,101

5.1.2. Lista rankingowa

Tablica 5.2: Wykonanie algorytmu *lista rankingowa*

Wielkość danych (GB)	Liczba maszyn	Runda	Czas (s)	Max zużycie pamięci (GB)	Max przesył danych (GB)
1	5	Próbkowanie	43	4,575	0,003
		Sortowanie	101	11,011	1,137
		Ranking	77	13,197	1,094
5	5	Próbkowanie	81	20,452	0,005
		Sortowanie	267	34,687	5,109
		Ranking	226	36,123	4,923
1	10	Próbkowanie	40	4,653	0,003
		Sortowanie	71	12,203	1,123
		Ranking	66	15,865	1,094
5	10	Próbkowanie	53	19,312	0,012
		Sortowanie	138	39,587	5,081
		Ranking	120	38,440	4,923

5.1.3. Statystyki prefiksowe

W algorytmie statystyk prefiksowych użyto sumy jako funkcji statystyk.

Tablica 5.3: Wykonanie algorytmu *statystyki prefiksowe*

Wielkość danych (GB)	Liczba maszyn	Runda	Czas (s)	Max zużycie pamięci (GB)	Max przesył danych (GB)
1	5	Próbkowanie	41	4,575	0,003
		Sortowanie	102	11,021	1,137
		Prefix	95	13,326	1,148
5	5	Próbkowanie	80	20,552	0,005
		Sortowanie	266	34,667	5,106
		Prefix	255	32,260	5,185
1	10	Próbkowanie	41	4,643	0,003
		Sortowanie	70	12,213	1,133
		Prefix	85	16,017	1,148
5	10	Próbkowanie	53	19,312	0,012
		Sortowanie	138	39,587	5,081
		Prefix	140	42,469	5,185

5.1.4. Grupowanie

W algorytmie grupowania użyto sumy jako funkcji statystyk.

Tablica 5.4: Wykonanie algorytmu *grupowania*

Wielkość danych (GB)	Liczba maszyn	Runda	Czas (s)	Max zużycie pamięci (GB)	Max przesył danych (GB)
1	5	Próbkowanie	41	4,555	0,003
		Sortowanie	101	11,011	1,117
		Grupowanie	49	8,899	0,0001
5	5	Próbkowanie	79	20,532	0,005
		Sortowanie	265	34,657	5,101
		Grupowanie	126	27,279	0,0006
1	10	Próbkowanie	40	4,623	0,003
		Sortowanie	71	12,211	1,113
		Grupowanie	51	10,842	0,0001
5	10	Próbkowanie	52	19,311	0,011
		Sortowanie	137	39,567	5,071
		Grupowanie	80	30,526	0,0001

5.1.5. Statystyka okienkowa

W algorytmie statystyki okienkowej użyto sumy jako funkcji statystyk.

Tablica 5.5: Wykonanie algorytmu *statystyki okienkowej*

Wielkość danych (GB)	Liczba maszyn	Runda	Czas (s)	Max zużycie pamięci (GB)	Max przesył danych (GB)
1	5	Próbkowanie	40	4,556	0,003
		Sortowanie	102	11,012	1,116
		Ranking	86	13,198	1,093
		Perf. sortowanie	94	13,994	1,168
		Stat. okienkowa	141	16,168	2,366
5	5	Próbkowanie	78	20,531	0,005
		Sortowanie	261	34,655	5,101
		Ranking	198	32,732	4,923
		Perf. sortowanie	257	40,410	5,255
		Stat. okienkowa	473	36,515	10,703
1	10	Próbkowanie	41	4,623	0,003
		Sortowanie	71	12,211	1,113
		Ranking	68	16,429	1,094
		Perf. sortowanie	82	17,788	1,168
		Stat. okienkowa	103	24,364	2,366
5	10	Próbkowanie	52	19,311	0,011
		Sortowanie	137	39,567	5,071
		Ranking	126	40,335	4,922
		Perf. sortowanie	155	46,564	5,254
		Stat. okienkowa	276	44,522	10,629

5.1.6. Podsumowanie

Tablica 5.6: Podsumowanie wykonania algorytmu *TeraSort* na Hadoopie.

Wielkość danych (GB)	Liczba maszyn	Czas (s)	Max zużycie pamięci (GB)	Max przesył danych (GB)
1	5	152	12,062	1,187
5	5	364	34,677	5,099
1	10	116	12,303	1,133
5	10	199	40,187	5,101

Tablica 5.7: Podsumowanie wykonania algorytmu *listy rankingowej* na Hadoopie.

Wielkość danych (GB)	Liczba maszyn	Czas (s)	Max zużycie pamięci (GB)	Max przesył danych (GB)
1	5	231	13,197	1,137
5	5	584	36,123	5,109
1	10	187	15,865	1,123
5	10	321	39,587	5,081

Tablica 5.8: Podsumowanie wykonania algorytmu *statystyk prefiksowych* na Hadoopie.

Wielkość danych (GB)	Liczba maszyn	Czas (s)	Max zużycie pamięci (GB)	Max przesył danych (GB)
1	5	248	13,326	1,148
5	5	611	34,667	5,185
1	10	206	16,017	1,148
5	10	341	42,469	5,185

Tablica 5.9: Podsumowanie wykonania algorytmu *grupowania* na Hadoopie.

Wielkość danych (GB)	Liczba maszyn	Czas (s)	Max zużycie pamięci (GB)	Max przesył danych (GB)
1	5	201	11,011	1,117
5	5	480	34,657	5,101
1	10	172	12,211	1,113
5	10	279	39,567	5,071

Tablica 5.10: Podsumowanie wykonania algorytmu *statystyk okienkowych* na Hadoopie.

Wielkość danych (GB)	Liczba maszyn	Czas (s)	Max zużycie pamięci (GB)	Max przesył danych (GB)
1	5	473	16,168	2,366
5	5	1277	40,410	10,703
1	10	375	24,364	2,366
5	10	756	46,564	10,629

Algorytmy *TeraSort*, *lista rankingowa*, *statystyki prefiksowe* oraz *grupowanie* mają bardzo zbliżone maksymalne zużycie pamięci oraz maksymalny przesył danych. Jedynie algorytm *statystyki okienkowej* zużywa więcej pamięci oraz transferu danych. Wynika to z faktu replikowania obiektów w ostatniej rundzie algorytmu.

Wszystkie algorytmy zakończyły się w skończonej liczbie rund (3 lub 5). Wbudowane narzędzie do monitorowania rund MapReduce nie obejmuje statystyk dotyczących zużytej pamięci i transferu danych na poszczególnych maszynach. Widać jednak, że maksymalne zużycie pamięci oraz maksymalny przesył danych na całym klastrze są ograniczone przez liczbę obiektów wejściowych. Dodatkowo czas, pamięć i transfer spadają liniowo wraz ze wzrostem ilości maszyn oraz wzrastają liniowo wraz ze wzrostem wielkości danych wejściowych. Na tej podstawie można stwierdzić, że implementacje przykładowych algorytmów minimalnych na Hadoopa przy użyciu biblioteki spełniają założenia klasy algorytmów minimalnych.

5.2. Spark

W podrozdziale zostały przedstawione wyniki testów poszczególnych algorytmów minimalnych zaimplementowanych przy użyciu biblioteki i uruchomionych na Sparku. Kolumny mają następujące znaczenie:

- Wielkość danych (GB) – wielkość danych wejściowych,
- Liczba maszyn – liczba maszyn DataNode użytych do wykonania algorytmu,
- Min / Max / Średni czas (s) – minimalny / maksymalny / średni czas wykonania rundy algorytmu na maszynie,
- Min / Max / Średnia pamięć (GB) – minimalne / maksymalne / średnie zużycie pamięci na maszynie,
- Min / Max / Średni przesył danych (GB) – minimalny / maksymalny / średni przesył danych na maszynie.

5.2.1. TeraSort

Tablica 5.11: Wykonanie algorytmu *TeraSort* na Sparku.

Wielkość danych (GB)	Liczba maszyn	Runda	Min czas (s)	Max czas (s)	Średni czas (s)	Min pamięć (GB)	Max pamięć (GB)	Średnia pamięć (GB)	Min przesył danych (GB)	Max przesył danych (GB)	Średni przesył danych (GB)
1	5	Sortowanie	14	17	16	0,221	0,225	0,224	0	0	0
5	5	Sortowanie	24	26	25	0,944	1,002	1,001	0	0	0
1	10	Sortowanie	11	15	13	0,108	0,112	0,112	0	0	0
5	10	Sortowanie	26	30	28	0,500	0,504	0,504	0	0	0

5.2.2. Lista rankingowa

Tablica 5.12: Wykonanie algorytmu *lista rankingowa* na Sparku.

Wielkość danych (GB)	Liczba maszyn	Runda	Min czas (s)	Max czas (s)	Średni czas (s)	Min pamięć (GB)	Max pamięć (GB)	Średnia pamięć (GB)	Min przesył danych (GB)	Max przesył danych (GB)	Średni przesył danych (GB)
		Sortowanie	14	17	16	0,221	0,225	0,224	0,000	0,000	0,000
1	5	Ranking	117	160	133	3,800	5,100	4,240	0,357	0,460	0,392
		Sortowanie	24	25	25	0,943	1,002	1,001	0,000	0,000	0,000
5	5	Ranking	720	740	733	18,600	20,500	19,840	1,671	1,849	1,760
		Sortowanie	11	15	13	0,108	0,113	0,112	0,000	0,000	0,000
1	10	Ranking	70	85	78	1,671	1,676	1,673	0,172	0,212	0,195
		Sortowanie	26	30	28	0,500	0,505	0,504	0,000	0,000	0,000
5	10	Ranking	280	315	302	9,000	10,600	9,890	0,815	0,950	0,882

5.2.3. Statystyki prefiksowe

W algorytmie statystyk prefiksowych użyto sumy jako funkcji statystyk.

Tablica 5.13: Wykonanie algorytmu *statystyki prefiksowe* na Sparku.

Wielkość danych (GB)	Liczba maszyn	Runda	Min czas (s)	Max czas (s)	Średni czas (s)	Min pamięć (GB)	Max pamięć (GB)	Średnia pamięć (GB)	Min przesył danych (GB)	Max przesył danych (GB)	Średni przesył danych (GB)
		Sortowanie	15	17	16	0,221	0,226	0,224	0,000	0,000	0,000
1	5	Prefix	125	157	138	3,400	5,100	4,240	0,337	0,443	0,390
		Sortowanie	24	25	25	0,954	1,002	1,001	0,000	0,000	0,000
5	5	Prefix	720	740	733	18,800	20,700	19,940	1,632	1,832	1,760
		Sortowanie	11	14	12	0,108	0,112	0,112	0,000	0,000	0,000
1	10	Prefix	70	77	74	1,661	1,673	1,668	0,174	0,212	0,195
		Sortowanie	24	25	24	0,500	0,504	0,504	0,000	0,000	0,000
5	10	Prefix	285	335	309	9,000	10,600	9,880	0,815	0,950	0,882

5.2.4. Grupowanie

W algorytmie grupowania użyto sumy jako funkcji statystyk.

Tablica 5.14: Wykonanie algorytmu *grupowania* na Sparku.

Wielkość danych (GB)	Liczba maszyn	Runda	Min czas (s)	Max czas (s)	Średni czas (s)	Min pamięć (GB)	Max pamięć (GB)	Średnia pamięć (GB)	Min przesył danych (GB)	Max przesył danych (GB)	Średni przesył danych (GB)
1	5	Sortowanie	15	17	16	0,221	0,225	0,224	0,000	0,000	0,000
		Grupowanie	65	85	75	3,100	5,500	4,200	0,274	0,470	0,372
5	5	Sortowanie	24	26	25	0,944	1,002	1,001	0,000	0,000	0,000
		Grupowanie	275	375	353	19,600	20,000	19,740	1,648	1,649	1,648
1	10	Sortowanie	11	15	13	0,108	0,112	0,112	0,000	0,000	0,000
		Grupowanie	39	42	41	1,662	1,668	1,664	0,183	0,184	0,183
5	10	Sortowanie	25	30	28	0,502	0,503	0,505	0,000	0,000	0,000
		Grupowanie	171	186	177	11,400	11,400	11,400	0,824	0,825	0,824

5.2.5. Statystyki okienkowe

W algorytmie statystyk okienkowych użyto sumy jako funkcji statystyk.

Tablica 5.15: Wykonanie algorytmu *statystyk okienkowych* na Sparku.

Wielkość danych (GB)	Liczba maszyn	Runda	Min czas (s)	Max czas (s)	Średni czas (s)	Min pamięć (GB)	Max pamięć (GB)	Średnia pamięć (GB)	Min przesył danych (GB)	Max przesył danych (GB)	Średni przesył danych (GB)
1	5	Sortowanie	15	17	16	0,221	0,225	0,224	0,000	0,000	0,000
		Ranking	124	158	136	3,800	5,100	4,300	0,377	0,453	0,411
		Perf. sortowanie	164	196	179	3,700	4,800	4,400	0,365	0,431	0,387
		Stat. okienkowa	64	68	65	3,500	3,500	3,500	0,726	0,726	0,726
5	10	Sortowanie	23	25	24	0,945	1,001	0,999	0,000	0,000	0,000
		Ranking	310	342	331	19,000	20,800	20,100	2,800	3,100	2,900
		Perf. sortowanie	439	470	455	23,300	26,800	24,800	3,500	3,700	3,700
		Stat. okienkowa	198	223	212	22,300	24,700	23,500	1,600	1,700	1,700
1	10	Sortowanie	11	15	13	0,108	0,111	0,112	0,000	0,000	0,000
		Ranking	75	85	81	1,672	1,672	1,672	0,188	0,204	0,199
		Perf. sortowanie	82	93	88	1,672	1,801	1,701	0,365	0,383	0,377
		Stat. okienkowa	48	50	49	3,400	3,500	3,500	0,362	0,362	0,362
5	10	Sortowanie	25	30	28	0,501	0,504	0,503	0,000	0,000	0,000
		Ranking	246	253	248	10,400	10,700	10,500	1,500	1,600	1,500
		Perf. sortowanie	280	294	288	11,500	11,700	11,600	1,800	1,900	1,800
		Stat. okienkowa	112	123	115	14,300	14,500	14,400	2,600	2,700	2,700

5.2.6. Podsumowanie

Tablica 5.16: Podsumowanie algorytmu *TeraSort* przy użyciu Sparka.

Wielkość danych (GB)	Liczba maszyn	Czas (s)	Max zużycie pamięci na poj. maszynie (GB)	Max przesył danych na poj. maszynie (GB)
1	5	41	0,225	0
5	5	66	1,002	0
1	10	37	0,112	0
5	10	52	0,504	0

Tablica 5.17: Podsumowanie algorytmu *lista rankingowa* na Sparku.

Wielkość danych (GB)	Liczba maszyn	Czas (s)	Max zużycie pamięci na poj. maszynie (GB)	Max przesył danych na poj. maszynie (GB)
1	5	192	5,100	0,460
5	5	780	20,500	1,849
1	10	115	1,676	0,212
5	10	360	10,600	0,950

Tablica 5.18: Podsumowanie algorytmu *statystyki prefiksowe* na Sparku.

Wielkość danych (GB)	Liczba maszyn	Czas (s)	Max zużycie pamięci na poj. maszynie (GB)	Max przesył danych na poj. maszynie (GB)
1	5	189	5,100	0,443
5	5	780	20,700	1,832
1	10	106	1,673	0,212
5	10	375	10,600	0,950

Tablica 5.19: Podsumowanie algorytmu *grupowania* na Sparku.

Wielkość danych (GB)	Liczba maszyn	Czas (s)	Max zużycie pamięci na poj. maszynie (GB)	Max przesył danych na poj. maszynie (GB)
1	5	117	5,500	0,470
5	5	416	20,000	1,649
1	10	72	1,668	0,184
5	10	231	11,400	0,825

Tablica 5.20: Podsumowanie algorytmu *statystyk okienkowych* na Sparku.

Wielkość danych (GB)	Liczba maszyn	Czas (s)	Max zużycie pamięci na poj. maszynie (GB)	Max przesył danych na poj. maszynie (GB)
1	5	454	5,100	0,726
5	5	1075	26,800	3,700
1	10	258	3,500	0,383
5	10	715	14,500	2,700

Algorytmy *lista rankingowa*, *statystyki prefiksowe* oraz *grupowanie* mają bardzo zbliżone maksymalne zużycie pamięci oraz maksymalny przesył danych. Algorytm *statystyk okienkowych*, na tle pozostałych algorytmów, zużywa więcej pamięci oraz transferu danych. Wynika to z faktu replikowania obiektów w ostatniej rundzie algorytmu.

Statystyki algorytmu *TeraSort* znacząco odstają od wyników reszty algorytmów. W bibliotece do implementacji algorytmów minimalnych został użyty wbudowany w Sparka algorytm sortowania i w związku z tym jest bardzo dobrze zoptymalizowany. Dodatkowo składa się z jednej rundy. Pozostałe algorytmy minimalne korzystają z wyniku sortowania i muszą zapisać go w pamięci RAM. Niestety maszyny wchodzące w skład klastra posiadały tylko 6 GB pamięci RAM i w związku z tym pomiędzy kolejnymi rundami algorytmu zapisywały dane do pamięci lokalnej maszyny. Skutkowało to zwiększeniem zużytej pamięci, a także wydłużeniem czasu wykonania.

Wszystkie algorytmy zakończyły się w skończonej liczbie rund. Maksymalne zużycie pamięci oraz maksymalny przesył danych są ograniczone przez liczbę obiektów wejściowych. Dodatkowo czas, pamięć i transfer spadają liniowo wraz ze wzrostem ilości maszyn oraz wzrastają liniowo wraz ze wzrostem wielkości danych wejściowych.

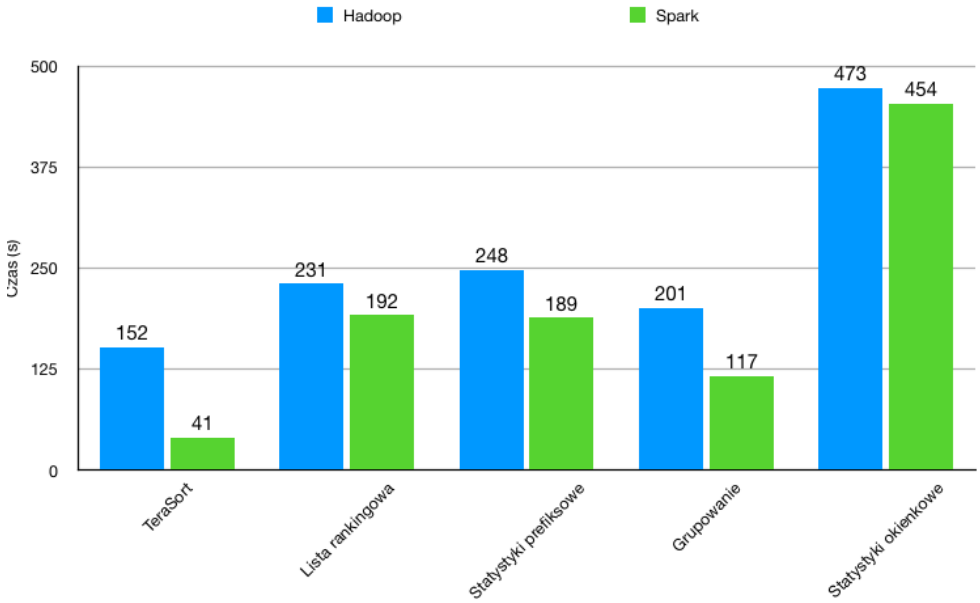
Warto także zauważyć, że wartości minimalne i maksymalne w obrębie czasów wykonania, zużycia pamięci oraz transferu danych nie są od siebie bardzo oddalone, co świadczy o równomiernym obciążeniu maszyn.

Na podstawie powyższych statystyk i analiz można stwierdzić, że implementacje przedstawionych algorytmów minimalnych na Sparka przy użyciu biblioteki spełniają założenia klasy algorytmów minimalnych.

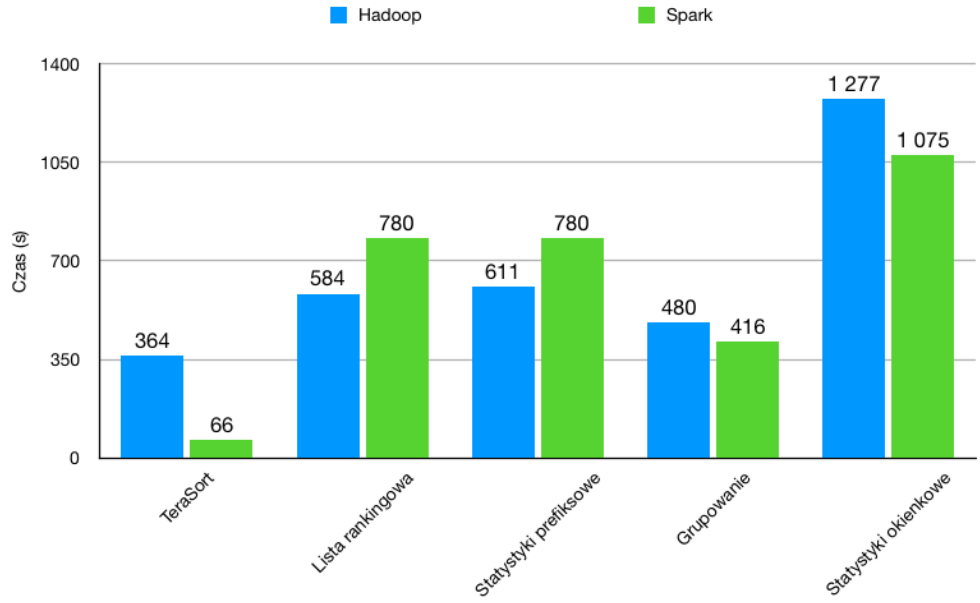
5.3. Porównanie Hadoop vs Spark

W podrozdziale porównano statystyki wykonania na Sparku i Hadoopie przykładowych algorytmów minimalnych zaimplementowanych przy użyciu biblioteki.

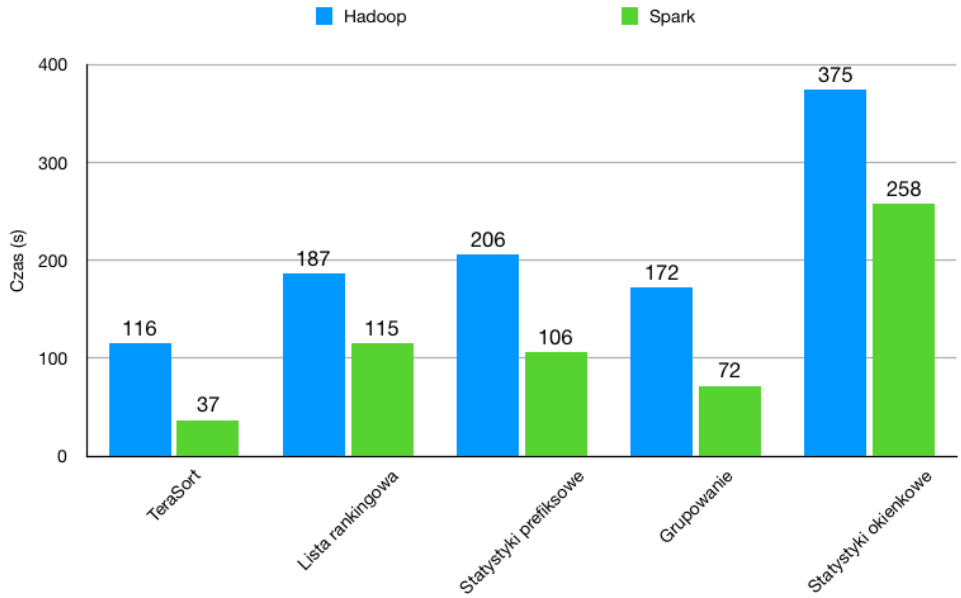
Rysunek 5.1: Porównanie czasów wykonania algorytmów minimalnych na 1 GB danych przy użyciu 5 maszyn.



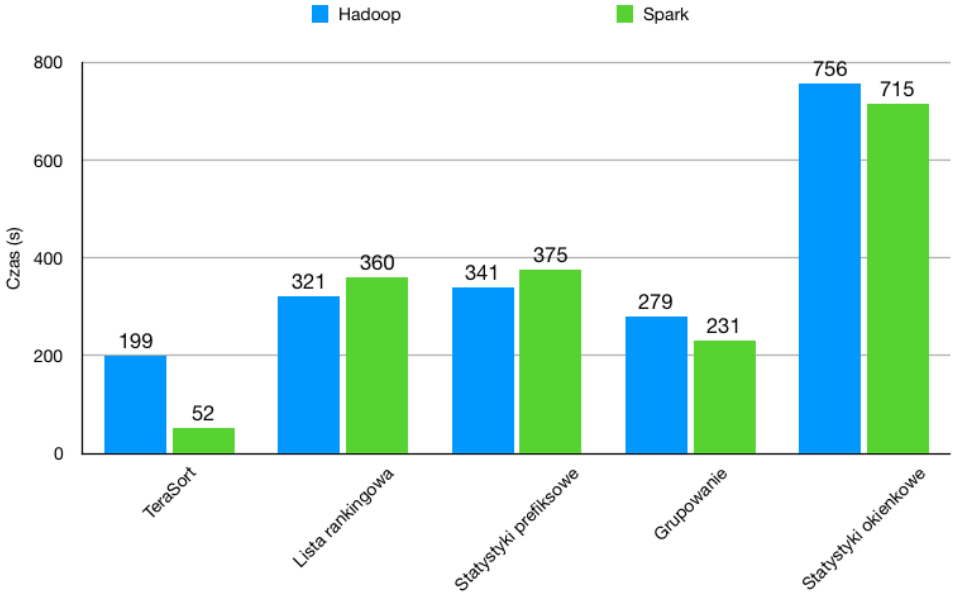
Rysunek 5.2: Porównanie czasów wykonania algorytmów minimalnych na 5 GB danych przy użyciu 5 maszyn.



Rysunek 5.3: Porównanie czasów wykonania algorytmów minimalnych na 1 GB danych przy użyciu 10 maszyn.



Rysunek 5.4: Porównanie czasów wykonania algorytmów minimalnych na 5 GB danych przy użyciu 10 maszyn.



Z wykresów na Rysunkach 5.1, 5.2, 5.3 oraz 5.4 wynika, że algorytm *TeraSort* jest znacząco szybszy na Sparku. W implementacji Sparka *grupowanie* oraz *statystyka okienkowa* są zawsze szybsze, jednak *lista rankingowa* oraz *statystyki prefiksowe* niekoniecznie. Możemy zauważyć, że wielkość danych wejściowych odgrywa bardzo ważną rolę. Przy rozmiarze danych 1 GB wszystkie algorytmy na Sparku są znacząco szybsze niż na Hadoopie. Natomiast przy wielkości danych 5 GB różnica maleje, a w przypadku *listy rankingowej* oraz *statystyk prefiksowych* Hadoop wygrywa. Powodem takiego zachowania jest zbyt mała ilość pamięci RAM na maszynach wchodzących w skład klastra. Jak wcześniej było wspomniane, Spark zawdzięcza swoją szybkość operacjom *in-memory*. Niestety w przypadku zbyt małej ilości pamięci RAM, ta przewaga zostaje zniwelowana, a dodatkowo system musi zapisywać i odczytywać dane z dysku, tym samym tracąc swoją najcenniejszą właściwość.

Rozdział 6

Podsumowanie

Biblioteka do implementacji algorytmów minimalnych powstała z myślą ułatwienia i przyspieszenia procesu tworzenia nowych algorytmów minimalnych oraz szybkiego dostępu do istniejących implementacji. W obecnych czasach programowanie rozproszone jest coraz powszechniej stosowane i dlatego uważam, że biblioteka idealnie wpasowuje się w aktualny trend pracy nad efektywnym przetwarzaniem ogromnych zbiorów informacji.

Obie biblioteki udostępniają szereg funkcjonalności takich jak wysyłanie obiektów na konkretne maszyny, równoległe operowanie na różnych typach danych oraz funkcje do obliczania łącznych statystyk na zbiorach.

Implementacja na Hadoopa wydaje się być dobrym narzędziem do używania, tworzenia i rozwijania algorytmów minimalnych. Posiada zwięzłe API oraz ułatwia korzystanie z paradygmatu MapReduce. Umożliwia także pisanie generycznych algorytmów MapReduce. Testy przykładowych algorytmów minimalnych wypadły bardzo obiecująco. Niepokojące może być wysokie zużycie pamięci lokalnej maszyn. W przyszłości warto by powtórzyć testy, mierząc statystyki oddzielnie dla każdej maszyny wchodzącej w skład klastra. Dodatkowo warto przeprowadzić testy w warunkach identycznych do opisanych w pracy [9], a następnie porównać wyniki.

Rozwiązanie na Sparka, pod kątem łatwości użytkowania wypada bardzo dobrze. Biblioteka jest intuicyjna w użyciu i komponuje się ze stylem programowania na Sparka. Oprócz wcześniej wymienionych funkcjonalności, umożliwia także pisanie generycznych algorytmów rozproszonych. Niestety w aspekcie efektywności, implementacja na Sparka prezentuje się nieco gorzej niż na Hadoopa. Z testów przeprowadzonych na przykładowych algorytmach minimalnych można wyciągnąć wniosek, że biblioteka nie jest do końca zoptymalizowana. Potwierdza to brak zdecydowanej przewagi Sparka nad Hadoopem pod względem efektywności. Dodatkowo fakt, że algorytm *TeraSort* zaimplementowany przy użyciu wbudowanego w Sparka algorytmu sortowania ma dużo lepsze statystyki niż reszta przykładowych algorytmów, pokazuje że biblioteka może zostać zoptymalizowana. Warto spojrzeć na *lineage* i uważnie przyjrzeć się kolejnym etapom wykonania algorytmu. Należałoby także powtórzyć testy na klastrze złożonym z maszyn o większej ilości pamięci RAM.

Obecna wersja biblioteki nie jest jeszcze perfekcyjna, jednak stanowi bazę do rozpoczęcia kolejnych prac. Implementacje przykładowych algorytmów spełniają warunki minimalności. Pozwala to optymistycznie patrzeć w przyszłość i zachęca do dalszego rozwoju biblioteki.

Bibliografia

[1] Avro description. https://avro.apache.org. Accessed: 2019-06-30.

[2] Protobuf description. https://developers.google.com/protocol-buffers. Accessed: 2019-06-30.

[3] Serialization benchmarks. https://labs.criteo.com/2017/05/serialization. Accessed: 2019-07-02.

[4] Spark documentation. https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html. Accessed: 2019-07-02.

[5] Thrift description. https://thrift.apache.org/docs. Accessed: 2019-06-30.

[6] Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, and Matei Zaharia. Scaling Spark in the real world: performance and usability. volume 8, pages 1840–1843. VLDB Endowment, August 2015.

[7] Telmo da Silva Morais. Survey on frameworks for distributed computing: Hadoop, Spark and Storm. In *Proceedings of the 10th Doctoral Symposium in Informatics Engineering-DSIE*, volume 15, 2015.

[8] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The Hadoop distributed file system. In *MSST*, volume 10, pages 1–10, 2010.

[9] Yufei Tao, Wenqing Lin, and Xiaokui Xiao. Minimal MapReduce algorithms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 529–540. ACM, 2013.

[10] Ronald C Taylor. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. In *BMC bioinformatics*, volume 11. BioMed Central, 2010.

[11] Hugh J Watson. Tutorial: Big data analytics: Concepts, technologies, and applications. In *Communications of the Association for Information Systems*, volume 34, page 65, 2014.

[12] Tom White. *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[13] Xindong Wu, Xingquan Zhu, Gong-Qing Wu, and Wei Ding. Data mining with big data. In *IEEE transactions on knowledge and data engineering*, volume 26, pages 97–107, 2013.

[14] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. volume 10, page 95. HotCloud, 2010.

[15] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache Spark: A unified engine for big data processing. In *Communications of the ACM*, volume 59, pages 56–65. ACM, 2016.