9.0

TRANSFORMACJA DANYCH W CZASIE RZECZYWISTYM: OCENA WYDAJNOŚCI APACHE SPARK

Mariusz Rafało
Szkoła Główna Handlowa w Warszawie

Emilia Kaleta-Pazdur
Autorka niezrzeszona

Streszczenie

Apache Spark to platforma do przetwarzania danych w trybie wsadowym oraz w czasie rzeczywistym. Ważne wyzwanie, zwłaszcza w trybie przesyłania w czasie rzeczywistym, stanowi zapewnienie wysokiej przepustowości i niskiego opóźnienia. Jest to szczególnie ważne w przypadku systemów AI działających w czasie rzeczywistym. W niniejszym rozdziale przedstawiamy serię eksperymentów mających na celu identyfikację parametrów i konfiguracji Apache Spark, które mogą zmniejszyć opóźnienia w transformacji danych. W badaniu zweryfikowaliśmy najpopularniejsze transformacje danych: grupowanie i filtrowanie. Kluczowe ustalenia wskazują, że wybór klastra jednowęzłowego może zapewnić korzystny kompromis i osiągnięcie równowagi między wydajnością przesyłania strumieniowego a efektywnością wykorzystania zasobów.

Słowa kluczowe: Apache Spark, wydajność przetwarzania, dane w czasie rzeczywistym

Rozdział z monografii: Doligalski T., Kaszyński D. (red. nauk.), Sztuczna inteligencja w przedsiębiorstwach i gospodarce (AI Spring 2024), Warszawa 2025.

Monografia oferowana jest w swobodnym dostępie do pobrania na stronie AI Lab SGH.

Wprowadzenie

Dane w czasie rzeczywistym odnoszą się do informacji, które są gromadzone, przetwarzane i wykorzystywane natychmiast po ich wygenerowaniu, umożliwiając działanie w oparciu na najnowszych informacjach. Technologia Apache Spark jest obecnie standardem w zakresie masowego przetwarzania danych. To platforma open source do równoległego przetwarzania danych w pamięci operacyjnej [Apache Spark, 2020b]. Apache Spark jest także używany w usługach wiodących dostawców chmurowych, takich jak Amazon EMR [Amazon, 2019], Databricks [Databricks, 2023] lub Google Dataproc [Google Cloud, 2022]. Spark oferuje ujednolicony system do obsługi dużych zbiorów danych, algorytmów uczenia maszynowego i możliwość przetwarzania danych w czasie rzeczywistym. Działa na jednym serwerze lub w klastrach złożonych z wielu węzłów (serwerów). Moduł Spark Streaming jest komponentem platformy Spark, służącym do przetwarzania danych w czasie rzeczywistym.

Inną technologią, powszechnie używaną do przesyłania strumieniowego danych, jest Apache Kafka. Podczas gdy Spark Streaming oferuje przetwarzanie danych i możliwości analityczne, Apache Kafka jest rozproszoną platformą przesyłania strumieniowego. Integracja obu platform upraszcza architekturę dla analiz w czasie rzeczywistym [Salloum, Dautov, Chen, Peng, Huang, 2016].

Spark działa w rozproszonym środowisku, w którym dane i zadania obliczeniowe są rozproszone na wielu serwerach [serwery w klastrze nazywa się potocznie węzłami (*nodes*)]. Architektura ta powoduje wyzwania w diagnozowaniu wąskich gardeł wydajności, ponieważ problemy mogą wynikać z różnych przyczyn, takich jak opóźnienie sieci, nierównomierna dystrybucja danych lub nieefektywna alokacja zasobów. Ponadto w Spark Streaming wyzwania te zostają spotęgowane przez potrzebę niskich opóźnień transformacji danych [Karau, Warren, 2017]. Zarządzanie przetwarzaniem w trybie strumieniowym wymaga utrzymywania i aktualizowania stanu danych w różnych partiach danych oraz różnych punktach czasu. Ponadto optymalizacja wykorzystania zasobów (takich jak pamięć i procesor) staje się bardziej złożona, ponieważ system musi znaleźć równowagę między przetwarzaniem o niskim opóźnieniu a ograniczeniami zasobów.

Co więcej, narzędzia diagnostyczne i metryki do monitorowania wydajności w Spark wymagają zrozumienia wewnętrznej architektury platformy. Przykładowo, prezentowane w Spark parametry diagnostyczne, takie jak wykorzystanie pamięci czy procesorów, nie dają jasnej wiedzy, w jaki sposób dane odczyty przekładają się na opóźnienia. Ten poziom wiedzy specjalistycznej nie zawsze jest łatwo dostępny.

Czynniki te sprawiają, że zarządzanie wydajnością w Spark, szczególnie w module Spark Streaming, to wymagające i złożone zadanie.

Celem rozdziału jest identyfikacja możliwości zwiększenia wydajności platformy Apache Spark w wybranych scenariuszach transformacji danych w czasie rzeczywistym. Przetwarzanie danych w czasie rzeczywistym ma duże znaczenie dla systemów sztucznej inteligencji, ponieważ modele AI mogą wówczas budować swoje predykcje w oparciu na najnowszych informacjach. Udostępnianie do modeli AI danych w czasie rzeczywistym pozwala na ciągłe douczanie modeli, uwzględniając nowe zdarzenia.

W badaniu skupiamy się wyłącznie na module Spark Streaming. Ponadto nie badamy wyłącznie technologii Spark, ale także jej integrację ze strumieniami Apache Kafka; kontrolujemy szybkość przepływu danych, co pozwala zweryfikować, w jaki sposób parametry Spark działają dla różnych szybkości transmisji danych.

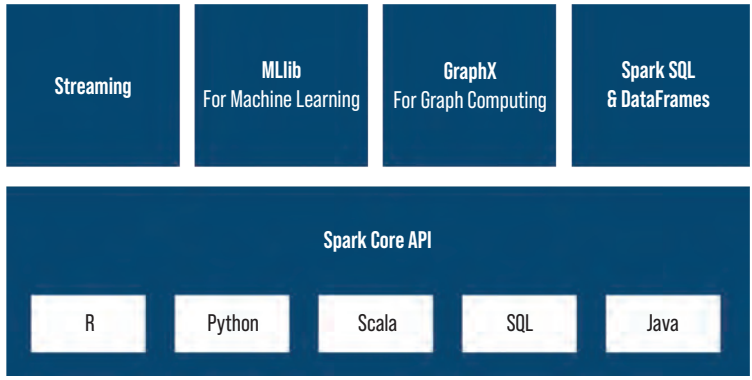
9.1. Technologia Apache Spark

Zwykle klaster Apache Spark składa się z serwera głównego (węzła) i wielu węzłów roboczych. W klastrze działają dwa typy procesów: sterownik (*driver*) i wykonawca (*executor*). Proces sterownika to główna jednostka przetwarzania danych; działa na węźle głównym i odpowiada za pozyskiwanie zasobów sprzętowych z klastra. Proces sterownika rozpoczyna się od przetłumaczenia aplikacji (kodu źródłowego) na zadania Spark. Następnie każde zadanie jest tłumaczone na logiczny plan wykonania, reprezentowany jako skierowany graf acykliczny (DAG). Wyjątek stanowią klastry jednowęzłowe, w których zarówno proces sterownika, jak i proces wykonawczy są uruchamiane na serwerze głównym. Każdy wykonawca ma wyłączne zasoby sprzętowe do obsługi zadań przetwarzania [Chao, Shi, Gao, Luo, Wang, 2018].

Transformacje danych w Apache Spark to procesy, które przekształcają zbiór danych z jednej formy w drugą. Reprezentacją struktury danych są w Spark tzw. ramki danych (*data frame*), które przypominają swoją budową tabelę (posiadają kolumny o określonych typach, zaś grupy kolumn są zorganizowane w wierszach).

Spark oferuje możliwość stosowania powszechnie używanych języków programowania, takich jak: Java, Python, SQL, R i Scala, oraz przetwarzania danych z różnych źródeł i w różnych formatach, takich jak płaskie pliki CSV, Parquet, Avro, ORC, JSON itp. Posiada kilka wbudowanych modułów (poza Spark Streaming), które służą do przetwarzania danych w zależności od celu, takich jak: Spark SQL przetwarzający dane strukturalne, MLib – biblioteka uczenia maszynowego czy GraphX przetwarzający grafy (rysunek 9.1).

Rysunek 9.1. Komponenty Apache Spark



Źródło: Damji, Wenig, Das, Lee [2020].

Informacje dotyczące głównych cech Apache Spark zostały zawarte w następujących stwierdzeniach [Apache Spark, 2020a]:

1. Dane w Spark są partycjonowane przy użyciu tzw. partycji haszujących. Partycjonowanie haszujące próbuje równomiernie rozłożyć dane na podstawie klucza [Chambers, Zaharia, 2018; Damji *et al.*, 2020].
2. Spark buforuje dane w pamięci operacyjnej na wielu węzłach.
3. Dzięki zastosowaniu ramek danych Spark zapewnia wysoką dostępność w przypadku awarii. Jeśli dany węzeł ulegnie awarii, Spark może ponownie obliczyć utraconą partycję z oryginalnego zestawu danych.
4. Spark jest wysoce skalowalny, z możliwością obsługi petabajtów (milionów gigabajtów) danych na wielu węzłach.

Zarządzanie wydajnością klastra Spark jest złożone ze względu na rozproszoną naturę Spark i potrzebę optymalizacji na różnych warstwach, w tym partycjonowania danych, serializacji i zarządzania pamięcią. Każda warstwa ma własny zestaw parametrów, które należy dostroić pod kątem zgodności z innymi. Spark oferuje ponad 180 parametrów konfiguracyjnych [Apache Spark, 2020a], z których kilkadziesiąt dotyczy dostrajania wydajności.

9.2. Badania powiązane

Badania wydajności Apache Spark skupiają się na dwóch strumieniach badawczych: ocenie wydajności [Wang, Khan, 2015] i przewidywaniu (predykcji) wydajności [Cheng, Ying, Wang, 2021]. Strumienie te jednak koncentrują się głównie na prze-

twarzaniu wsadowym. Strumień badania wydajności Spark dotyczy porównywania wydajności technologii przy wykorzystaniu testów porównawczych i dostrajania parametrów Spark. Popularnymi scenariuszami testów porównawczych wydajności są WordCount i Terasort [IBM, 2021]. Na przykład Ahmed, Barczak, Rashid i Susnjak [2021] wykonali testy oparte na tych scenariuszach dla 18 parametrów Spark. Kluczowym odkryciem było to, że Spark ma wyższą wydajność w porównaniu z technologią Hadoop MapReduce. Spark okazał się szybszy dwukrotnie w teście WordCount i czternastokrotnie w przypadku TeraSort. Naukowcy wykorzystują również kompleksową platformę testową HiBench [Ahmed *et al.*, 2021] – zunifikowaną do oceny wydajności systemów intensywnie wykorzystujących dane. Samadi, Zbakh i Tadonki [2018] przeprowadzili porównanie technologii Hadoop MapReduce z technologią Spark, korzystając z tej platformy. Okazało się, że Spark odznacza się większym wykorzystaniem procesora i operacji wejścia/wyjścia, a jednocześnie wyższą wydajnością niż MapReduce (szczególnie w przypadku obsługi większej ilości danych). W tym nurcie badawczym istnieją również badania bezpośrednio skupiające się na wydajności algorytmów AI działających na Spark [Mavridis, Karatza, 2017].

Nurt badawczy związany z predykcją wydajności Spark jest np. reprezentowany przez Petridisa, Gounarisa i Torresa [2017]. Badacze wskazują, w jaki sposób zastąpić domyślne wartości parametrów wartościami oszacowanymi przez modele predykcyjne. Badanie koncentrowało się głównie na algorytmach sortowania, kompresji i zarządzaniu pamięcią. Dowiedziono w nim, iż największy wpływ na jakość predykcji wydajności Apache Spark ma parametr *spark.shuffle.compress*.

Inne podejście do predykcji i dostrajania parametrów związanych z wydajnością zostało wprowadzone przez Petrova, Butakova, Nasonova i Melnika [2018]. W artykule zaproponowano adaptacyjny model wydajności, dedykowany skalowaniu zasobów systemowych w zależności od wymagań dotyczących opóźnień. Przeprowadzono zestaw eksperymentów, który pozwolił na uzyskanie 66% poprawy wydajności.

9.3. Zakres eksperymentu

Eksperyment polegał na uruchomieniu serii testów przetwarzania Spark Streaming zintegrowanych z Apache Kafka. Zadania Spark obejmowały połączenie z określonym strumieniem Kafka – tematem (*topic*), a następnie zebranie danych i przeprowadzenie transformacji. Eksperyment został przygotowany w języku programowania Python. Kody źródłowe umieszczono w publicznym repozytorium Github. Utworzono dwa programy: program producenta, który wysyłał dane do tematu Kafka z określoną

prędkością, oraz program konsumenta, który odczytywał dane z tematu, wykonywał odpowiednie transformacje i zapisywał wyniki. Przygotowaliśmy szereg scenariuszy testowych – każdy z nich obejmuje połączenie Apache Spark z tematem Kafka, pobranie przychodzących danych i ich transformację. Scenariusze zostały zdefiniowane przez:

- konfigurację prędkości transmisji danych (600, 1200, 1800 i 3000 rekordów na minutę);
- konfigurację liczby partycji w temacie Kafka (1, 2 lub 3);
- konfigurację sprzętową klastra Apache Spark (liczba węzłów);
- typ transformacji danych (filtrowanie i grupowanie);
- ustawienia parametrów Spark.

Wszystkie scenariusze testowe zostały uruchomione w infrastrukturze usług Amazon EC2. Platforma Apache Spark została dostarczona przez środowisko Databricks. Wykorzystaliśmy Structured Streaming API w Apache Spark. Każdy z wykonawców w klastrach miał 4 rdzenie procesora i 30,5 GB pamięci RAM.

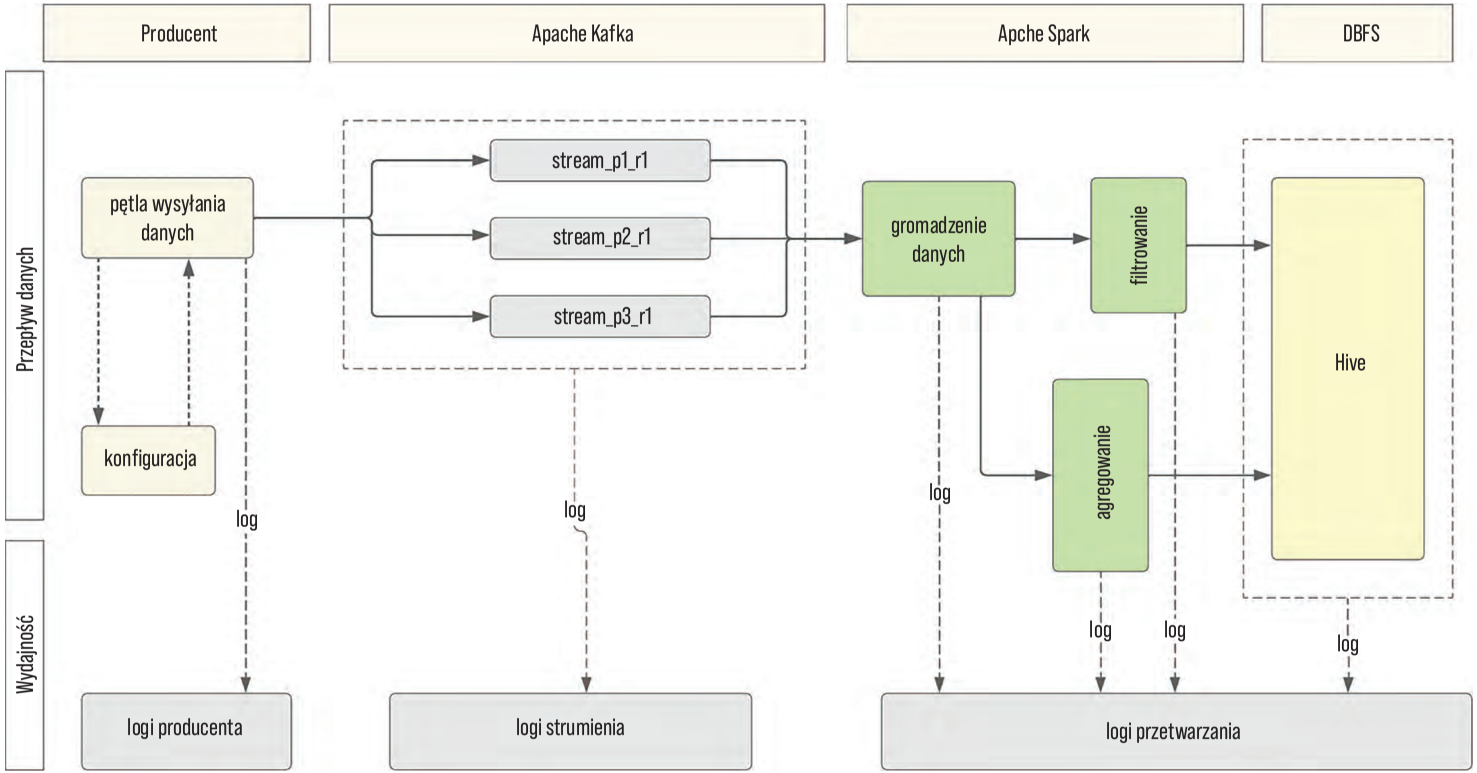
9.3.1. Przepływ danych

Koncepcję eksperymentu i przepływ danych przedstawiono na rysunku 9.2. Producent generuje dane w pętli, na podstawie konfiguracji zapisanej w pliku. Po stronie Spark konsument łączy się online ze strumieniem danych, po czym odczytuje dane z określonego tematu bezpośrednio do ramki danych. Następnie dane są transformowane.

Na każdym etapie transmisji i transformacji danych konsument rejestruje informacje, zawierające znacznik czasu na danym etapie. Posiadając dane znacznika czasu na każdym etapie przetwarzania danych, może zaś obliczyć opóźnienia (latencje) dla każdego kroku. Następnie obliczone opóźnienie jest zapisywane w repozytorium plików, wraz z konfiguracją Spark i Kafka oraz stanem infrastruktury sprzętowej klastra Spark.

Podczas badania przeanalizowaliśmy wszystkie parametry Apache Spark, aby wybrać te, które (zgodnie z dokumentacją) mogą mieć wpływ na wydajność strumienia danych. Następnie podczas oceny zidentyfikowaliśmy parametry, mające największy wpływ na wydajność przetwarzania danych.

Rysunek 9.2 Koncepcja przepływu danych w eksperymentach



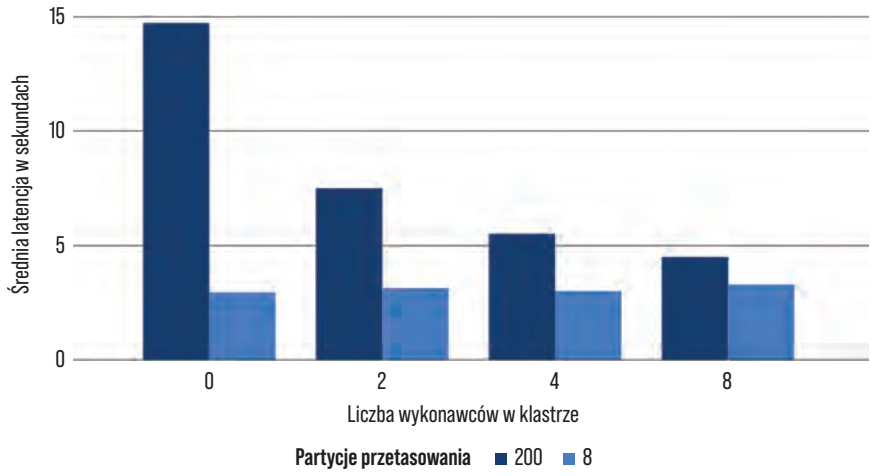
Źródło: opracowanie własne.

9.3.2. Ocena eksperymentu

Eksperyment został przeprowadzony przy stałej prędkości danych podczas danego testu. Z tego powodu utrzymaliśmy domyślną konfigurację Spark dla parametrów: maxOffsetsPerTrigger i minOffsetsPerTrigger, które w obu przypadkach były ustawione na wartość none. Wskaźnikiem wydajności była najniższa latencja, ale braliśmy również pod uwagę efektywne wykorzystanie zasobów.

Skonfigurowaliśmy skrypt inicjalizacyjny, który kontroluje częstotliwość zbierania logów. Następnie w aplikacji Spark ustawiliśmy parametr konfiguracyjny: spark.sql.streaming.metricsEnabled na true, aby włączyć gromadzenie logów. Logi metryk były zapisywane w pięciosekundowych odstępach. Biorąc pod uwagę wszystkie kombinacje scenariuszy, warianty prędkości transmisji i parametry, obliczyliśmy, że cały eksperyment trwał ponad 40 godzin.

Rysunek 9.3. Średnia latencja dla grupowania przy różnych ustawieniach parametru liczby partycji przetasowania danych



Źródło: opracowanie własne.

Rysunek 9.3 przedstawia wyniki dla scenariuszy z grupowaniem z różną konfiguracją parametru liczby partycji przetasowania danych (*shuffle partitions*): 200 (domyślnie) lub 8. Przetasowanie danych jest istotne dla transformacji takich jak grupowanie i oznacza, że uzyskanie ostatecznego wyniku wymaga przemieszania ich między partycjami Spark. Natomiast transformacje takie jak filtrowanie są wykonywane bez przetasowania danych, dlatego zmiana konfiguracji tego parametru nie wpływa na testy z filtrowaniem. Wykres pokazuje znaczący spadek latencji przy zmniejszonej liczbie

partycji przetasowania danych. Spadki latencji stają się mniejsze wraz ze wzrostem liczby wykonawców w klastrze. Niemniej jednak najniższa średnia latencja występuje dla klastra jednowęzłowego przy ustawieniu ośmiu partycji przetasowania danych. Dodatkowo poddaliśmy analizie wielkość próby i odchylenie standardowe dla scenariuszy przedstawionych na rysunku 9.3. Dane zebrane w tabeli 9.1 pokazują relatywnie duże liczności prób, przy przeważająco niskich odchyleniach standardowych. Biorąc pod uwagę tę analizę, wykluczyliśmy z dalszych testów dane oparte na domyślnej konfiguracji liczby partycji przetasowania danych (200) ze względu na wyższe odchylenie standardowe.

Tabela 9.1. Liczebność prób i odchylenie standardowe dla danych zaprezentowanych na rysunku 9.3

Liczba wykonawców	Partycje przetasowania	Liczebność próby	Odchylenie standardowe
0	8	2123	1,40
0	200	3463	3,66
2	8	2132	0,52
2	200	2046	2,25
4	8	2309	0,82
4	200	2266	1,91
8	8	2128	0,42
8	200	2104	0,84

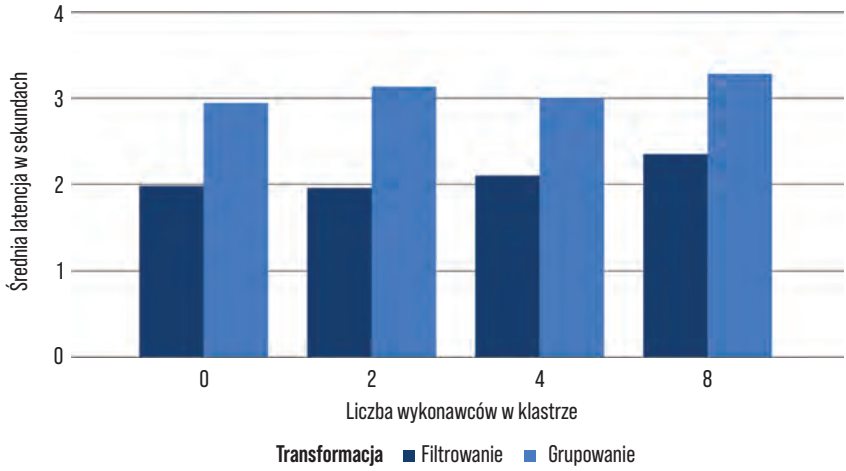
Źródło: opracowanie własne.

Rysunek 9.4 przedstawia średnią latencję dla obu transformacji danych w zależności od liczby wykonawców w klastrze. Dla każdej wielkości klastra średnia latencja jest konsekwentnie niższa przy wykonywaniu filtrowania w porównaniu z grupowaniem. Jest to związane z dwoma czynnikami. Po pierwsze, grupowanie jest szeroką transformacją i wymaga przetasowania danych. Po drugie, grupowanie jest również transformacją z pamięcią stanu (*stateful transformation*), co oznacza, że pomiędzy kolejnymi wsadami danych przechowywany jest ich stan w pamięci. Wykres wskazuje również, że najlepsza średnia latencja występuje dla klastra jednowęzłowego przy grupowaniu oraz dla klastra jednowęzłowego i klastra z dwoma wykonawcami przy filtrowaniu. Dla obu transformacji największy klaster ma najwyższą średnią latencję.

Rysunek 9.4 sugeruje również, że klaster jednowęzłowy może stanowić wystarczającą infrastrukturę dla testowanych scenariuszy. Aby potwierdzić to założenie, potrzebna jest bardziej szczegółowa analiza danych. Dlatego po analizie średniej latencji dla

transformacji danych dla danej wielkości klastra zbadaliśmy średnią latencję dla każdego testu. Poszczególne testy różnią się od siebie na podstawie różnych konfiguracji strumienia, takich jak rozmiar wsadu danych (*batch size*) lub liczba partycji Kafki.

Rysunek 9.4. Średnia latencja dla grupowania i filtrowania



Źródło: opracowanie własne.

Tabela 9.2. Najlepsza i najgorsza latencja dla każdego scenariusza

Transformacja	Prędkość przepływu danych	Najlepsza średnia latencja		Najgorsza średnia latencja	
		liczba wykonawców	średnia latencja w sekundach	liczba wykonawców	średnia latencja w sekundach
Filtrowanie	600	4	1,72	8	10,19
Filtrowanie	1200	2	1,81	4	4,16
Filtrowanie	1800	0	1,79	4	5,05
Filtrowanie	3000	4	1,60	8	5,17
Grupowanie	600	4	2,34	2	4,24
Grupowanie	1200	4	2,66	4	3,51
Grupowanie	1800	0	2,59	4	3,89
Grupowanie	3000	4	2,43	0	3,60

Źródło: opracowanie własne.

Wybór najlepszych i najgorszych wyników dla każdego scenariusza przedstawiono w tabeli 9.2. Średnia latencja dla optymalnej konfiguracji waha się od 1,5 do 2,7 sekundy, a dla najgorszej konfiguracji od 3,5 do nawet 10 sekund. Dla dwóch

z ośmiu scenariuszy (oba przy prędkości przepływu danych 1800) klaster jednowęzłowy wykazuje najlepszą średnią latencję.

Dla pozostałych scenariuszy (dla prędkości 600, 1200 i 3000) przeanalizowaliśmy najlepszą średnią latencję dla pojedynczego testu przeprowadzonego na klastrze jednowęzłowym, aby sprawdzić, jak dokonać jego porównania z klastrem, który uzyskał najlepsze rezultaty. Tabela 9.3 przedstawia porównanie najlepszej średniej latencji dla pojedynczego testu wykonanego na wygrywającym klastrze z najlepszą średnią latencją dla pojedynczego testu przeprowadzonego na klastrze jednowęzłowym – obydwa dla tego samego scenariusza.

Ostatnia kolumna w tabeli 9.3 pokazuje różnicę między średnią latencją najlepszego testu wykonanego na klastrze jednowęzłowym a średnią latencją testu przeprowadzonego na wygrywającym klastrze. Różnica jest nieistotna dla połowy scenariuszy, a dla pozostałych wynosi mniej niż 500 milisekund. W dalszym ciągu jest to zgodne z naszym założeniem, aby unikać nadmiernego rozbudowywania infrastruktury.

Tabela 9.3. Najlepsza średnia latencja dla klastra wygrywającego oraz dla klastra jednowęzłowego

Transformacja	Prędkość przepływu danych	Liczba wykonawców	Najlepsza średnia latencja w sekundach dla wygrywającego klastra	Najlepsza średnia latencja w sekundach dla klastra jednowęzłowego	Różnica w sekundach
Filtrowanie	600	4	1,72	1,96	0,24
Filtrowanie	1200	2	1,81	1,87	0,06
Filtrowanie	1800	0	1,79	nie dotyczy	nie dotyczy
Filtrowanie	3000	4	1,60	1,93	0,33
Grupowanie	600	4	2,34	2,60	0,26
Grupowanie	1200	4	2,66	2,67	0,01
Grupowanie	1800	0	2,59	nie dotyczy	nie dotyczy
Grupowanie	3000	4	2,43	2,48	0,05

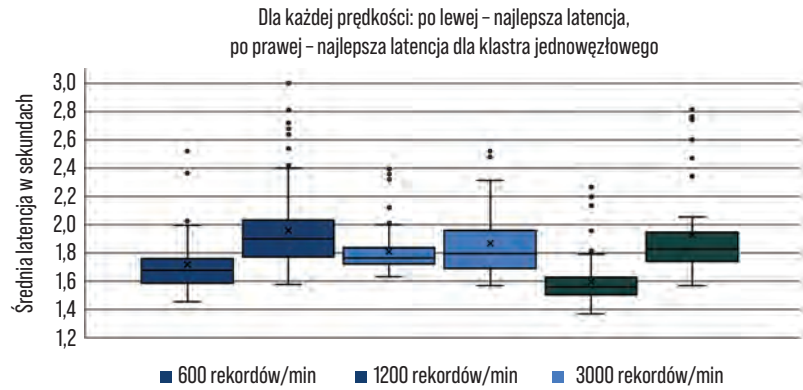
Źródło: opracowanie własne.

W kolejnym kroku poddaliśmy analizie rozkład danych dotyczących latencji dla każdego testu. Rysunki 9.5 i 9.6 pokazują rozkład latencji odpowiednio dla scenariuszy z filtrowaniem i grupowaniem. Każdy rysunek obejmuje tylko te scenariusze, dla których klaster jednowęzłowy nie osiągnął najlepszego rezultatu (scenariusze dla prędkości: 600, 1200 i 3000 rekordów na minutę). Dla każdej prędkości przepływu danych przedstawiono dwa wykresy pudełkowe. Wykres po lewej stronie przedstawia rozkład danych dla testu przeprowadzonego na wygrywającym klastrze, a po prawej –

rozkład latencji dla testu przeprowadzonego na najlepiej skonfigurowanym klastrze jednowęzłowym. Znak „x" oznacza średnią latencję.

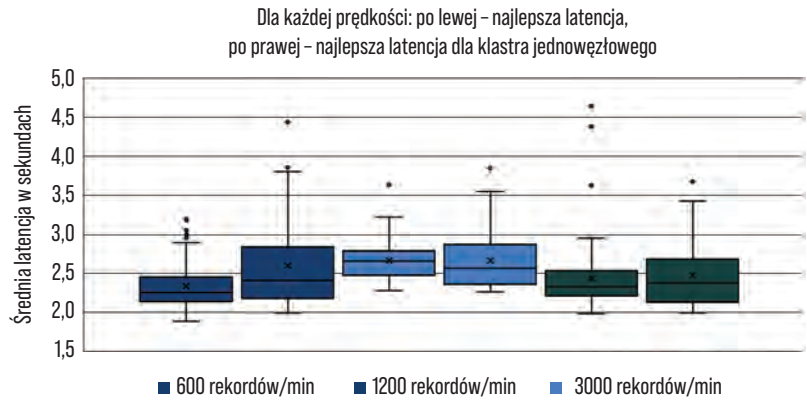
Testy przeprowadzone na klastrze jednowęzłowym wykazują szerszy rozkład danych, co może wskazywać, że infrastruktura jest mniej stabilna. Jednak w przypadku scenariuszy z filtrowaniem 75% danych nadal oscyluje poniżej 2 sekund. Co więcej, w przypadku dwóch scenariuszy z grupowaniem trzeci kwartyl rozkładu jest tylko nieznacznie wyższy od tych dla wygrywających klastrów.

Rysunek 9.5. Rozkład danych najlepszej średniej latencji dla filtrowania – zestawienie najlepszej średniej latencji ogółem z najlepszą średnią latencją dla klastra jednowęzłowego dla danej prędkości



Źródło: opracowanie własne.

Rysunek 9.6. Rozkład danych najlepszej średniej latencji dla grupowania – zestawienie najlepszej średniej latencji ogółem z najlepszą średnią latencją dla klastra jednowęzłowego dla danej prędkości

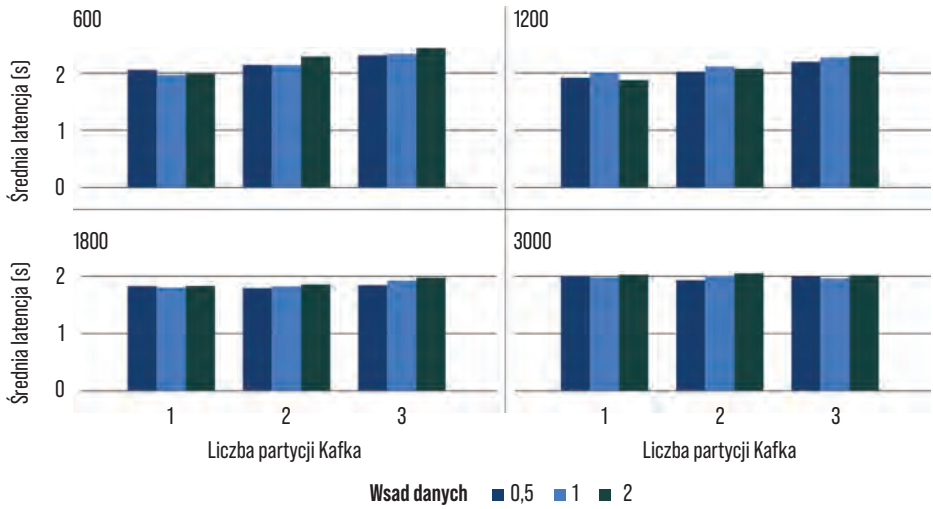


Źródło: opracowanie własne.

Najgorszy wynik odnotowano dla scenariusza z grupowaniem i najniższą prędkością przepływu danych: trzeci kwartyl latencji zmienia się z 2,45 do 2,84 sekundy. W przypadku, gdy stała wydajność nie jest priorytetem, wyniki klastra jednowęzłowego mogą być zadowalające, a wybór mniejszego klastra może okazać się opłacalną strategią.

Rysunki 9.7 i 9.8 przedstawiają po cztery wykresy odpowiednio dla transformacji filtrowania i grupowania. Każdy z nich odpowiada innej prędkości przepływu danych. Z kolei każdy słupek przedstawia średnią latencję w zależności od wielkości wsadu danych po stronie konsumenta strumienia danych i liczby partycji Kafki. Wielkość wsadu danych (0,5 s, 1 s lub 2 s) wyzwala mikro-wsad w regularnych odstępach czasowych i następuje przetwarzanie przyjętych danych we wsadzie. Ponadto po stronie producenta strumienia danych ustawiliśmy liczbę partycji Kafki.

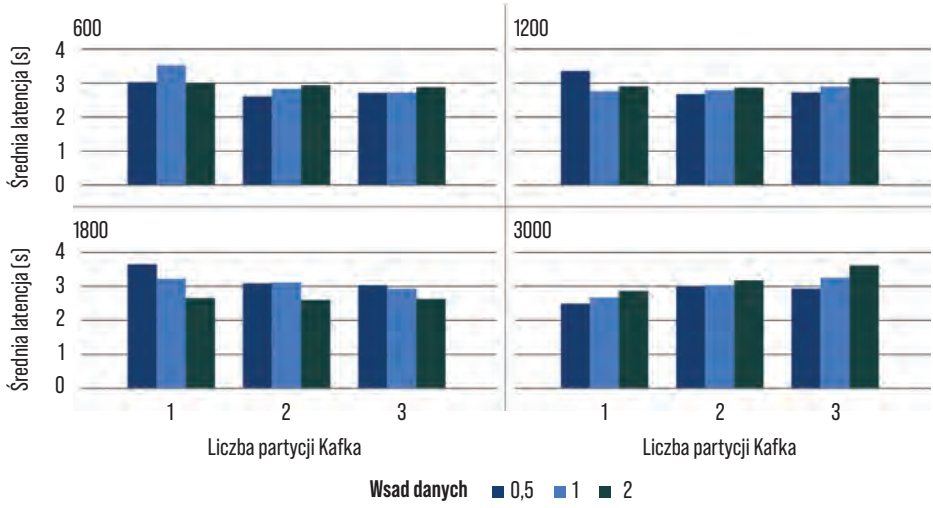
Rysunek 9.7. Średnia latencja dla filtrowania dla klastra jednowęzłowego



Źródło: opracowanie własne.

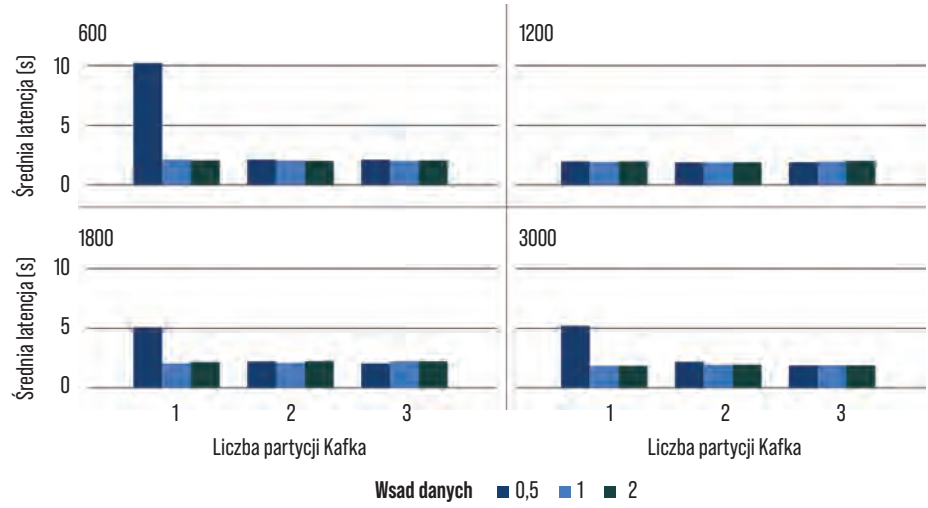
Rysunek 9.7 wskazuje, że w przypadku filtrowania latencja jest najniższa, gdy strumienie odczytują dane z tematu Kafki z jedną partycją. Tendencja ta stopniowo maleje przy wyższych prędkościach przepływu danych. Przy prędkości 3000 rekordów na minutę nie zaobserwowano znaczących zmian w latencji w różnych konfiguracjach. Natomiast w przypadku scenariuszy z grupowaniem zmiany latencji nie są tak spójne pomiędzy różnymi prędkościami danych. Najwyraźniejszy trend jest widoczny przy najwyższej prędkości. Zmniejszenie liczby partycji Kafki i zmniejszenie rozmiaru wsadu danych po stronie konsumenta prowadzi do spadku latencji. Pojedyncza zmiana ustawień może zmniejszyć latencję nawet o 25%.

Rysunek 9.8. Średnia latencja dla grupowania dla klastra jednowęzłowego



Źródło: opracowanie własne.

Rysunek 9.9. Średnia latencja dla filtrowania dla klastra z 8



Źródło: opracowanie własne.

Dodatkowo rysunek 9.9 przedstawia interesującą obserwację dla klastra z ośmioma wykonawcami w scenariuszach z filtrowaniem. Wskazuje on, że średnia latencja dla strumieni danych o wielkości wsadu danych co 500 milisekund oraz przy najniższej prędkości danych gwałtownie wzrasta do 10 sekund. Odpowiednia mediana laten-

cji wynosi 4 sekundy, ale wciąż jest dwa razy wyższa niż mediana latencji dla innych rozmiarów wsadu przy tej samej prędkości danych. Jest to najwyższa średnia latencja zaobserwowana w badaniu. W przypadku rozdzielania mniejszej liczby rekordów strumieniowych na większą liczbę węzłów może pojawić się znaczący narzut operacyjny.

Podsumowanie

W niniejszym artykule opisano przetestowanie różnych scenariuszy przetwarzania danych przy użyciu różnych ustawień konfiguracyjnych. Przede wszystkim wyniki wskazują, że zmniejszenie liczby partycji przetasowania danych podczas grupowania może znacznie poprawić wydajność zapytań. Ponadto transformacje bez pamięci stanu okazały się mniej czasochłonne niż transformacje z pamięcią stanu. Średnio klaster z ośmioma wykonawcami wypadł najgorzej. W poszczególnych testach klaster jednowęzłowy osiągnął najniższą średnią latencję dla niektórych scenariuszy przy optymalnej konfiguracji. Dla innych scenariuszy wykazano, że średnia latencja jest wyższa maksymalnie o 330 milisekund od latencji dla wygrywającego klastra. Te analizy wskazują, że wybór lżejszej infrastruktury może być opłacalny, nawet jeśli wiąże się to z większymi wahaniami latencji. Biorąc pod uwagę wyniki dla klastra jednowęzłowego, zaleca się również zbadanie różnych wartości zarówno dla wielkości wsadu danych, jak i liczby partycji Kafki. Czynniki te mogą wpływać na latencję.

Wyniki badania mają na celu pomóc projektantom architektury w podejmowaniu optymalnych decyzji. Dostarczają wskazówek dotyczących wyboru odpowiedniej wielkości klastra Apache Spark i ustawień parametrów. Co więcej, zapewniają wgląd w możliwe skutki różnych scenariuszy strumieniowania. Optymalizując alokację zasobów i zadania przetwarzania w Apache Spark, organizacje mogą znacząco zredukować zużycie zasobów obliczeniowych oraz czas potrzebny na wnioskowanie, co prowadzi do obniżenia kosztów operacyjnych. Ponadto Spark dostrojony pod kątem wydajności zapewnia elastyczność w tworzeniu i wdrażaniu modeli AI, wykorzystujących dane w czasie rzeczywistym.

Rozważając wyniki tego badania, należy wziąć pod uwagę pewne ograniczenia. Po pierwsze, w naszym badaniu uwzględniliśmy dwa rodzaje transformacji: grupowanie i filtrowanie. Inne transformacje, takie jak liczenie unikalnych wartości czy wykrywanie anomalii, z pewnością byłyby wartościowe. Po drugie, skupiliśmy się wyłącznie na technologiach Apache Spark i Apache Kafka. Ciekawe byłoby porównanie tych technologii z innymi technologiami (np. systemami opartymi na chmurze).

Kierunki dalszych badań wynikają z ograniczeń tego badania. Uwzględnienie bardziej złożonych transformacji i analiz lub rozszerzenie zakresu analizowanych technologii z pewnością zwiększyłoby praktyczną wartość badania.

Bibliografia

Ahmed, N., Barczak, A.L.C., Rashid, M.A., Susnjak, T. (2021). An Enhanced Parallelisation Model for Performance Prediction of Apache Spark on a Multinode Hadoop Cluster, *Big Data and Cognitive Computing*, 5(4). DOI: 10.3390/bdcc5040065.

Amazon (2019). *Amazon EMR*, https://aws.amazon.com/emr/ (dostęp: 18.10.2019).

Apache Spark (2020a). *Spark Configuration*, https://spark.apache.org/docs/latest/configuration.html (dostęp: 14.05.2021).

Apache Spark (2020b). *Unified Engine for Large-Scale Data Analytics*, https://spark.apache.org/ (dostęp: 22.07.2021).

Chambers, B., Zaharia, M. (2018). *Spark: The Definitive Guide Big Data Processing Made Simple*, https://learning.oreilly.com/library/view/spark-the-definitive/9781491912201/ (dostęp: 10.09.2022).

Chao, Z., Shi, S., Gao, H., Luo, J., Wang, H. (2018). A Gray-Box Performance Model for Apache Spark, *Future Generation Computer Systems*, *89*, s. 58–67. DOI: 10.1016/j.future.2018.06.032.

Cheng, G., Ying, S., Wang, B. (2021). Tuning Configuration of Apache Spark on Public clouds by Combining Multi-Objective Optimization and Performance Prediction Model, *Journal of Systems and Software*, *180*, 111028. DOI: 10.1016/j.jss.2021.111028.

Damji, J.S., Wenig, B., Das, T., Lee, D. (2020). Learning Spark: Lightning-Fast Data Analytics. W: *Learning Spark: Lightning-Fast Data Analytics* (2nd ed.). Sebastopol: O'Reilly Media.

Databricks (2023). *Apache Spark™*, https://www.databricks.com/spark/about (dostęp: 23.10.2023).

Google Cloud (2022). *Dataproc*, https://cloud.google.com/dataproc (dostęp: 12.10.2022).

IBM (2021). *TeraSort Benchmark*, https://www.ibm.com/docs/en/spectrum-symphony/7.2.1?topic=mapreduce-terasort-benchmark (dostęp: 22.10.2022).

Karau, H., Warren, R. (2017). *High Performance Spark*. Sebastopol: O'Reilly Media.

Mavridis, I., Karatza, H. (2017). Performance Evaluation of Cloud-Based Log File Analysis with Apache Hadoop and Apache Spark, *Journal of Systems and Software*, 125, s. 133–151. DOI: 10.1016/j.jss.2016.11.037.

Petridis, P., Gounaris, A., Torres, J. (2017). Spark Parameter Tuning via Trial-and-Error, *Advances in Intelligent Systems and Computing*, 529, s. 226–237. DOI: 10.1007/978-3-319-47898.

Petrov, M., Butakov, N., Nasonov, D., Melnik, M. (2018). Adaptive Performance Model for Dynamic Scaling Apache Spark Streaming, *Procedia Computer Science*, 136, s. 109–117. DOI: 10.1016/j.procs.2018.08.243.

Salloum, S., Dautov, R., Chen, X., Peng, P.X., Huang, J.Z. (2016). Big Data Analytics on Apache Spark, *International Journal of Data Science and Analytics*, 1(3–4), s. 145–164. DOI: 10.1007/s41060-016-0027-9.

Samadi, Y., Zbakh, M., Tadonki, C. (2018). Performance Comparison Between Hadoop and Spark Frameworks Using HiBench Benchmarks, *Concurrency and Computation: Practice and Experience*, 30(12). DOI: 10.1002/cpe.4367.

Wang, K., Khan, M.M.H. (2015). Performance Prediction for Apache Spark Platform. W: *Proceedings – 2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security and 2015 IEEE 12th International Conference on Embedded Software and Systems* (s. 166–173). DOI: 10.1109/HPCC-CSS-ICESS.2015.246.