



AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ INŻYNIERII METALI I INFORMATYKI PRZEMYSŁOWEJ

KATEDRA INFORMATYKI STOSOWANEJ I MODELOWANIA

Praca dyplomowa magisterska

Rozproszony system zarządzania danymi dla pojazdów autonomicznych oparty o technologie Big Data.

Distributed data management system for autonomous vehicles based on the Big Data technologies.

Autor:

Piotr Tutak

Kierunek studiów:

Informatyka Techniczna

Opiekun pracy:

dr hab. Piotr Maciął

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

*Dziękuję Promotorowi
za ofiarowany mi czas
i cenne rady.*

Spis treści

Wstęp	7
Cel pracy	8
1 Problemy przetwarzania dużych zbiorów danych	11
1.1 Analiza przypadku użycia.	11
1.2 Niezawodność	13
1.2.1 Awarie sprzętowe	13
1.2.2 Błędy oprogramowania	14
1.2.3 Błędy ludzkie	14
1.3 Skalowalność	15
1.3.1 Obciążenie	15
1.3.2 Wydajność	15
1.3.3 Skalowanie	15
1.4 Łatwość w utrzymaniu	16
1.4.1 Operacyjność	16
1.4.2 Prostota	17
1.4.3 Ewolucyjność	17
1.5 Zastosowanie technik Big Data	18
2 Modele architektury Big Data	19
2.1 Podstawowe rodzaje przetwarzania	19
2.2 Architektura Lambda	20
2.2.1 Warstwa wsadowa	20
2.2.2 Warstwa obsługująca	21
2.2.3 Warstwa szybka	21
2.2.4 Wady i zalety architektury Lambda	21
2.3 Architektura Kappa	22
2.3.1 Wady i zalety architektury Kappa	23
2.4 Podsumowanie	24
3 Narzędzia Big Data	25
3.1 Hadoop	25
3.1.1 Model Map Reduce	25

3.1.2	HDFS - Hadoop Distributed File System	26
3.1.3	YARN - Yet Another Resource Negotiator	26
3.2	Spark	26
3.2.1	Resilient Distributed Dataset	27
3.2.2	DataFrame i SQL	28
3.2.3	Spark Streaming	28
3.3	Storm	28
3.4	Flink	29
3.5	Kafka	30
3.5.1	Kafka Streams	30
3.6	RabbitMQ	31
3.7	HBase i Apache Phoenix	31
3.8	Podsumowanie i wybór narzędzi	32
3.9	Narzędzia pomocnicze	33
3.9.1	Zookeeper	33
3.9.2	Ansible	33
4	Projekt i funkcje systemu	34
4.1	Wymagania	34
4.2	Funkcje systemu	35
4.3	Projekt rozwiązania	36
4.4	Model danych	39
4.4.1	Strumień wejściowy	39
4.4.2	Zdefiniowane tabele	40
4.4.3	Strumienie wyjściowe	40
5	Realizacja	41
5.1	Infrastruktura	41
5.1.1	Szczegółowa realizacja rozwiązania	41
5.2	Orkiestracja	46
5.2.1	Realizacja modelu danych	46
5.2.2	REST API	48
5.2.3	Symulacja powstawania zbioru danych	51
6	Prezentacja wyników	55
6.1	Inicjalizacja	55
6.2	Stworzenie zbioru danych	58
6.3	Stworzenie tabel	62
6.4	Stworzenie strumieni wyjściowych	64
7	Podsumowanie i wnioski	69

Wstęp

Ilość generowanych danych każdego roku nieustannie rośnie. Zgodnie z najnowszymi statystykami rynek Big Data był warty w 2018 ok. 170 miliardów dolarów. Do 2022 roku wartość rynku skoncentrowanego na przetwarzaniu dużych zbiorów danych ma osiągnąć oszałamiającą wartość 275 miliardów dolarów, przy założeniu 13% wzrostu rok do roku.[4]

Również ilość generowanych danych jest imponująca. Według informacji przekazanych przez World Economic Forum [5] do 2020 roku całkowita ilość danych wygenerowanych w internecie sięgnie wartości 44 zettabajtów. Jeden zettabajt to 1000^7 bajtów, czyli całkowita ilość danych to inaczej 44 tryliony gigabajtów. To około 40 razy więcej niż jest gwiazd w obserwowalnym wszechświecie.

Codziennie generowanych jest 500 milionów tweetów, 294 miliardy emaili, 4 petabajty (1000^5 B) danych na Facebook'u, 4 terabajty danych jest generowanych przez pojedyńczy samochód, 65 miliardów wiadomości jest wysyłanych poprzez aplikację WhatsApp. Dokonywanych jest 5 miliardów wyszukiwań dziennie.

Do 2025 roku zakłada się że dziennie będziemy generować ok. 470 exabajtów (1000^6 B) danych, co odpowiada ilościowo pojemności ok. 213 000 000 nośników DVD.

Tak duża ilość danych stawia przed ludzkością, a przed technologiami informatycznymi w szczególności, nowe wyzwania. Dotyczą one przede wszystkim sposobu na przechowywanie i przetwarzanie tak dużych zbiorów danych.

W odpowiedzi na narastające potrzeby wynikające z coraz większej ilości generowanych danych zaczęto tworzyć oprogramowanie, które jest w stanie sprostać tego typu wyzwaniom. W 2004 roku Google zaprezentowało algorytm MapReduce zaprojektowany dla systemów rozproszonych mający znacznie przyspieszyć operacje na dużych zbiorach danych przechowywanych w systemach rozproszonych [7]. W 2006 roku oficjalnie powstał otwartoźródłowy projekt Hadoop, który został stworzony na bazie projektu o nazwie Nutch, opisującym sposób rozproszonego składowania danych. Następnie zaczęły pojawiać się kolejne, nowe dystrybucje i projekty opensource ułatwiające i jeszcze bardziej przyspieszające operacje na dużych zbiorach danych.

Pojawienie się nowych możliwości przetwarzania danych powoduje też coraz szybszy przyrost generowanych danych. Potrzeby i rozwiązania generują nowe potrzeby, rozwiązanie aktualnych potrzeb umożliwia stworzenie nowych, wcześniej niemożliwych do zaspokojenia potrzeb.

Wraz z rozwojem Big Data mamy do czynienia z rozwojem AI, sztucznej inteligencji, której technologie w dużej mierze opierają się na możliwości przetwarzania dużych zbiorów danych. By stworzyć dobry model, który będzie właściwie podejmował decyzje, trzeba najpierw mieć

odpowiednio duży zbiór danych. Jak powszechnie wiadomo, modele stworzone przy użyciu technik uczenia maszynowego nie posiadają zdolności ekstrapolacji poza te dane, które wykorzystaliśmy do ich nauczenia. Dlatego im większy zbiór danych wejściowych, tym większa szansa, że model będzie działał w każdej napotkanej sytuacji.

By pozostać konkurencyjnym coraz więcej firm zwraca się w stronę AI [4]. AI zajmuje się rekomendowaniem produktów na Facebooku, przewidywaniem dróg jakimi poruszają się klienci w supermarketach, tworzeniem profili psychologicznych, przewidywaniem najbardziej optymalnych operacji giełdowych, rozpoznawaniem mowy czy rozpoznawaniem obrazów.

Jednym z problemów, na który szczególnie zwraca uwagę niniejsza praca, a który staje się ostatnimi czasy bardzo popularny, jest próba stworzenia pojazdów autonomicznych.

Cel pracy

Rozwój pojazdów autonomicznych jest doskonały przykładem połączenia Big Data ze sztuczną inteligencją. By móc tworzyć takie pojazdy, potrzebny jest duży zbiór danych zbieranych z pojazdów wyposażonych we wszelkiego rodzaju czujniki. Dzięki powiększaniu się zbioru danych, można zacząć trenować modele, które usprawniają i czynią pojazdy coraz bardziej autonomicznymi. Tak opisywany model tworzenia pojazdów nie jest możliwy bez pozyskania dużej ilości danych.

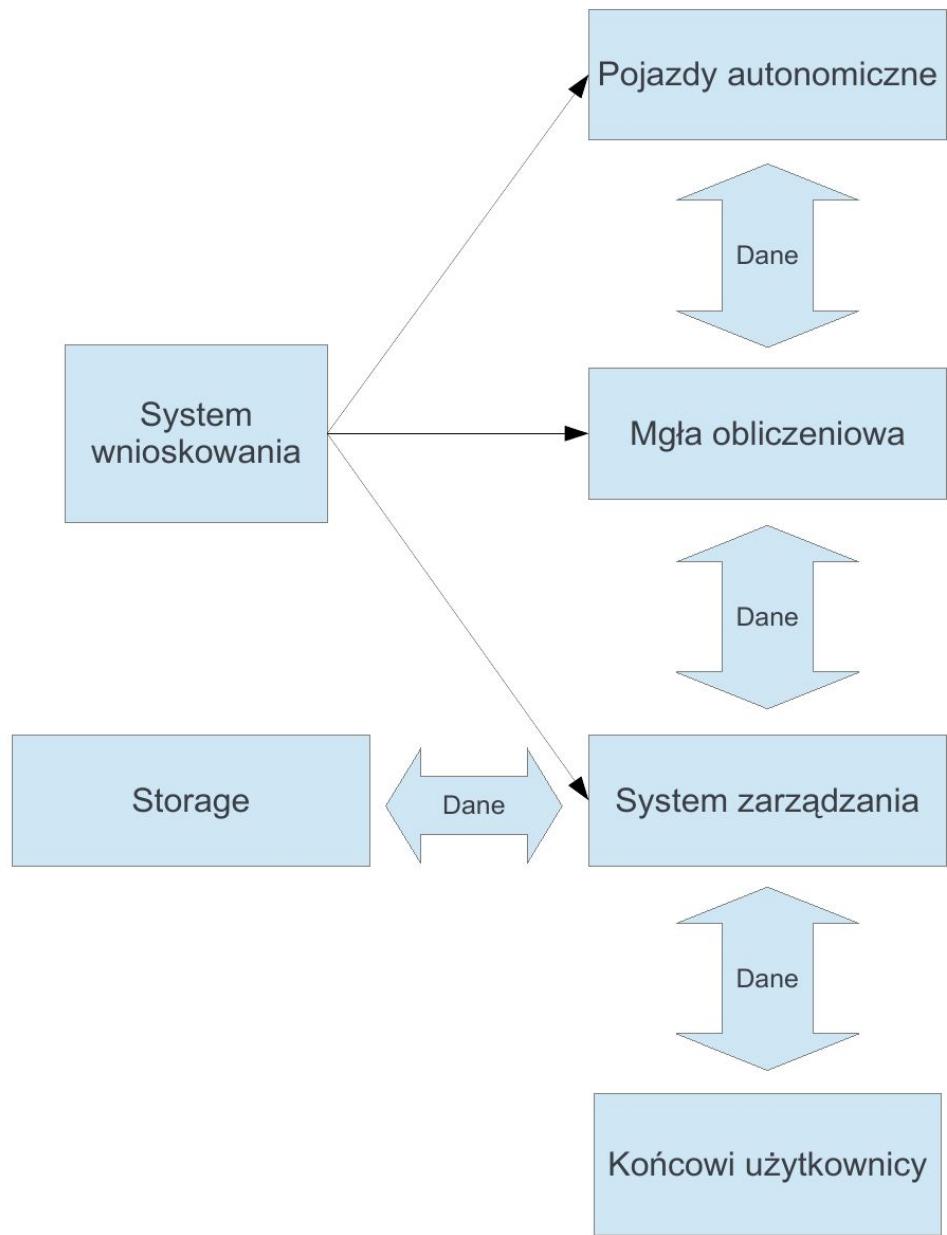
Celem niniejszej pracy będzie stworzenie sprawnego i elastycznego systemu zarządzania danymi pochodzącyimi z pojazdów autonomicznych przy pomocy narzędzi Big Data, umożliwiającym przetwarzanie danych w celu uczenia nowych modeli wspomagających podejmowanie decyzji, jak i wspomagających samo podejmowanie decyzji poprzez umożliwienie analizy danych w locie.

W ogólnym założeniu cały system składałby się z kilku podstawowych składowych, to jest z pojazdów autonomicznych, mgły obliczeniowej, chmury obliczeniowej, system zarządzania, systemu wnioskowania oraz końcowego użytkownika. Ogólny, uproszczony schemat takiego systemu jest widoczny na rysunku nr 1.

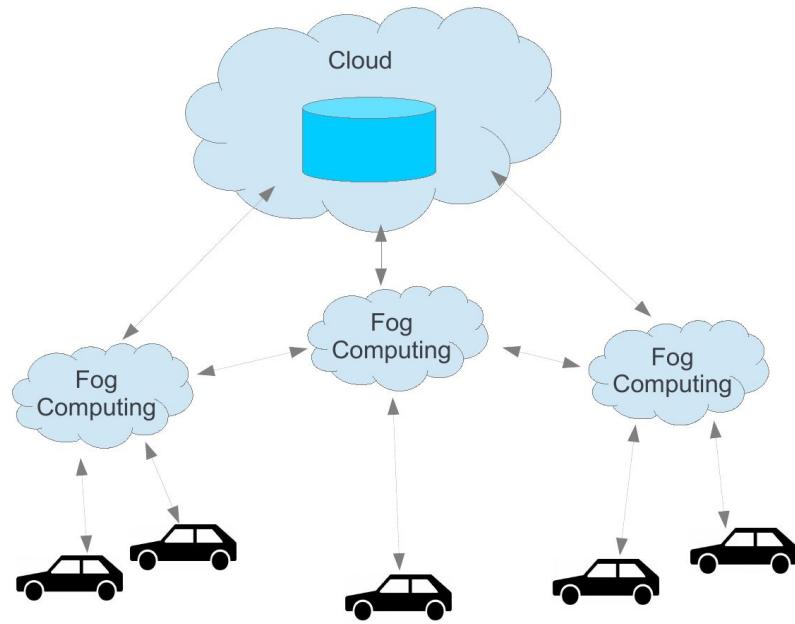
System zarządzania ma służyć koordynowaniu przepływu danych pomiędzy chmurą obliczeniową (Rys. 2), która z kolei zbiera dane z pojazdów autonomicznych i wstępnie je przetwarza, a końcowym użytkownikiem, którym w tym wypadku byłaby osoba, lub też program, mający na celu wyświetlanie statystyk, czy też uzyskanie danych służących trenowaniu modeli opartych o uczenie maszynowe, które miałyby zostać później użyte w samych pojazdach autonomicznych.

System zarządzania ma też być odpowiedzialny za przesyłanie danych do systemu przechowywania danych, jak i za pobieranie danych z tego systemu i udostępnianie ich zarówno końcowemu użytkownikowi jak i systemowi wnioskowania.

By stworzyć dany system w niniejszej pracy zostaną najpierw przedstawione pewne podstawy dotyczące przetwarzania informacji, następnie zostaną zaprezentowane stosowane typy architektury, których używa się do rozwiązania tego typu problemów. W kolejnym kroku zo-



Rysunek 1: Schemat połączonych systemów służących do zarządzania danymi dla pojazdów autonomicznych.



Rysunek 2: Cloud computing.

staną opisane i porównane narzędzia za pomocą których realizowany będzie system zarządzania. W końcowej części pracy zostaną przedstawione pewne szczegóły implementacji wraz z opisem.

Rozdział 1

Problemy przetwarzania dużych zbiorów danych

Jak zostało wspomniane we wstępie ilość danych generowanych jest bardzo duża. Jeden pojazd autonomiczny w ciągu dnia generuje 4 TB (terabajty) danych, czyli 4 tysiące gigabajtów. Należy założyć, że system zarządzania danych nie będzie przetwarzał danych pochodzących tylko z jednego pojazdu, ale z bardzo wielu pojazdów. Obsługiwanie tak dużych zbiorów danych wiąże się z pewnymi problemami, które zostaną w tym rozdziale omówione.

1.1 Analiza przypadku użycia.

By zaprezentować pewne problemy przetwarzania danych warto posłużyć się prostym przykładem [12]. Zadaniem jest stworzenie systemu, który będzie służył do zliczania zapytań dowolnej strony zadanej przez klienta. Dodatkowo system ma umożliwiać wyświetlanie 100 najczęściej odwiedzanych stron. By rozwiązać ten problem tworzony jest system bazodanowy z jedną tabelą i serwerem, do którego można wysłać sygnał identyfikujący jedno odwiedzenie obserwowanej strony internetowej. W momencie w którym dany sygnał dociera do serwera serwer zwiększa licznik odwiedzin w tabeli dla danej strony.

W przypadku, gdy obciążenie jest małe takie rozwiązanie spełnia swoje zadanie. Jednakże

Column name	Type
id	integer
user_id	integer
url	varchar(255)
pageviews	bigint

Rysunek 1.1: Przykładowa tabela systemu służącego do zliczania wejść na stronę [12].

okazuje się, że obciążenie, czyli liczba odwiedzin na śledzonych stronach może być tak duża, że baza danych nie potrafi poradzić sobie z uaktualnianiem cały czas tego samego pola za każdym kliknięciem. By rozwiązać ten problem, zamiast uaktualniać bazę danych za każdym razem, gdy dana strona zostanie odwiedzona, można zacząć uaktualniać ją dopiero gdy liczba odwiedzin osiągnie pewien poziom, np. 100. Wtedy licznik będzie zwiększany raz na sto razy. Zostaje stworzona zatem kolejka odwiedzin oraz odpowiedni proces, który odczytuje liczbę odwiedzin z kolejki i co 100 odwiedzin uaktualnia bazę danych.

Okazuje się jednak, że liczba odwiedzin jest tak duża, że także to rozwiązanie jest niewystarczające i po czasie problem pojawi się na nowo.

Kolejnym krokiem jest tzw. skalowanie horyzontalne, zwane też shardingiem. Wpisy w bazie danych, czyli poszczególne strony, zostają rozdzielone równomiernie na poszczególne shardy bazy danych, każdy shard odpowiada za odrębną część bazy danych, co sprawia równomierne rozłożenie obciążenia bazy.

Wiąże się to jednak ze znacznie zwiększoną złożonością tego bardzo prostego problemu. Należy dokonać reshardingu istniejącej bazy. Również wyświetlanie 100 najczęściej odwiedzanych stron jest bardziej skomplikowane, gdyż należy teraz scalać wyniki z poszczególnych shardów. Gdy aplikacja zliczająca staje się coraz bardziej popularna potrzebne są kolejne reshadingi, które zajmują coraz więcej czasu. Cała ta praca wymaga stworzenia specjalnych skryptów automatyzujących dany proces, który co jakiś czas trzeba powtarzać. Nie trudno też o jakiś błąd przy błędym przypisaniu liczby zliczeń dla poszczególnej strony przy procesie reshardingu.

W końcu system jest tak rozbudowany i równolegle działających maszyn jest tak dużo, że któraś z nich może ulec awarii. By temu zaradzić można wprowadzić kopię zapasową dla każdego shardu. Trzeba również wprowadzić opcję powiadamiania, że niektóre shardy bazy danych są niedostępne.

Mожет się okazać, że w kodzie produkcyjnym został odkryty błąd i liczba odwiedzin była zwiększana o 2 zamiast o 1 przy każdym sygnale wysyłanym do serwera. Błąd spostrzeżony dopiero po 24h zdążył już wyrządzić nieodwracalne szkody w bazie danych, gdyż został również rozprowadzony na wszystkie kopie zapasowe.

Powyższy przykład pokazuje pewne problemy, które są napotykane w przypadku przetwarzania dużej ilości danych nawet wtedy, gdy problem sam z siebie, taki jak zwiększanie pewnego licznika, wydaje się być bardzo prosty.

Poszczególne problemy, które narzędzia Big Data są w stanie rozwiązać, a które możemy wyróżnić analizując ten przykład to:

- Niezawodność
- Skalowalność
- Łatwość w utrzymaniu

1.2 Niezawodność

Jednym z pierwszych wyzwań do rozwiązywania przez tworzone oprogramowanie jest problem niezawodności [9]. Niezawodna aplikacja powinna spełniać pewne ogólnie przyjęte i zrozumiałe kryteria:

- Aplikacja wykonuje funkcję do której została przeznaczona.
- Aplikacja jest w stanie poradzić sobie z błędami użytkowników czy też z próbą użycia jej w niewłaściwy sposób.
- Aplikacja działa na tyle dobrze by poradzić sobie z przewidywanym dla niej obciążeniem i ilością danych, które musi przetworzyć.
- System jest w stanie zapobiec nieautoryzowanemu dostępowi do danych i wykorzystaniu ich w niewłaściwym celu.

Aplikację niezawodną zatem można by określić jako działającą dobrze nawet pomimo niesprzyjających okoliczności. Program spełniający takie kryteria możemy też nazwać jako odporny na awarie. Nigdy nie da się uniknąć wszystkich awarii przez dany system, ale można stworzyć oprogramowanie tak, by było odporne na najczęściej spotykane problemy.

Często by zweryfikować, że dany system jest rzeczywiście odporny na awarie, poddaje się go celowym awariom by sprawdzić, że wszelkie możliwe zabezpieczenia działają zgodnie ze swoim przeznaczeniem.

Najczęściej spotykanymi błędami, czy też awariami są:

- Awarie sprzętowe.
- Błędy oprogramowania.
- Błędy ludzkie.

1.2.1 Awarie sprzętowe

Średni czas życia dysku twardego bez awarii to ok. 10 do 50 lat [9]. W przypadku zatem, gdy mamy w naszym systemie 10000 dysków, awaria dysku będzie się zdarzała średnio codziennie.

Najczęstszym i najprostszym sposobem na radzenie sobie z awariami sprzętu jest nadmiarowość. Tworzy się dodatkowe systemy awaryjnego zasilania, awaryjne urządzenia, które mogą przejąć funkcję urządzeń, które działają w razie gdyby te uległy awarii.

W przypadku jednak systemów, których złożoność rośnie w bardzo dużym tempie nadmiarowość sprzętowa może być niewystarczająca. Zamiast niej, zaczyna się tworzyć aplikacje, które są w stanie poradzić sobie w sytuacji, gdy któryś komponent wpięty do systemu przestaje działać. Takie działanie osiąga się przede wszystkim poprzez wprowadzenie nadmiarowości dzięki technikom programowym. Dodatkowo takie systemy mają możliwość radzenia sobie w prosty sposób z aktualizacjami, gdyż można aktualizować poszczególne części systemu bez przerw w działaniu.

1.2.2 Błędy oprogramowania

Błędy oprogramowania to błędy systemowe. Są spowodowane przez źle zaprojektowane oprogramowanie. Niektóre z przykładów takich błędów to:

- Wyścig do zasobów, w którym biorą udział poszczególne części systemu.
- Część systemu przestaje odpowiadać lub zaczyna zwracać błędne dane.
- Kaskadowo postępujący błąd programu powodujący lawinę błędów we wszystkich częściach systemu.

Najczęściej tego typu błędy wynikają z pewnych wstępnych założeń odnośnie środowiska w jakim działa dany system. W początkowej fazie działania programu te założenia były prawdziwe, jednakże po pewnym czasie środowisko uległo zmianie i spowodowało fałszywość założeń przyjętych na początku działania programu i tym samym doprowadziło do awarii.

Na tego typu błędy nie ma łatwego i prostego rozwiązania. Najczęściej stosowanymi są przede wszystkim:

- Dogłębne i szerokie testy oprogramowania.
- Testowanie środowiska.
- Monitorowanie.
- Analiza zachowań systemu w środowisku produkcyjnym.

1.2.3 Błędy ludzkie

Okazuje się, że najczęstszym źródłem błędów działania systemu jest człowiek [12].

Najczęstszym sposobem radzenia sobie z błędami ludzkimi są:

- Tworzenie systemów i interfejsów w taki sposób by uprzykrzały życie, jeśli człowiek chciałby popełnić błąd i ułatwiały niezrobienie błędu. Okazuje się jednak, że przy zbyt restrykcyjnych ograniczeniach użytkownicy lubią je óbchodzić”, dlatego konieczne jest tutaj odpowiednie wyważenie.
- Stworzenie systemu *sandbox*, który umożliwia testowanie zachowania systemu w odizolowanym środowisku od środowiska produkcyjnego.
- Dogłębne testowanie od testów jednostkowych, poprzez testy integracyjne, aż po testy manualne.
- Stworzenie funkcji szybkiego przywracania systemu, który mógłby paść ofiarą błędu ludzkiego.
- Stworzenie oprogramowania monitorującego poszczególne części systemu, które może alarmować szybko o tym, że któraś część systemu nie działa prawidłowo.
- Dobre praktyki zarządzania i szkolenia.

1.3 Skalowalność

Kolejnym problemem jest skalowalność. Skalowalność to zdolność systemu do obsłużenia coraz większego obciążenia [9]. Najczęściej rozumie się ją, jako pewne właściwości systemu, które sprawiają, że można go łatwo rozbudowywać, co później przekłada się na możliwości działania pod coraz bardziej zwiększającym się obciążeniem.

Podstawowymi parametrami, które są pomocne w określeniu tego, czy system jest skalowalny, są obciążenie i wydajność.

1.3.1 Obciążenie

Obciążeniem nazywamy ten zbiór parametrów, które są najbardziej istotne dla systemu. Nie można powiedzieć, że istnieje z góry określona lista takich parametrów. Wszystko zależy od tego jakie jest przeznaczenie naszego systemu. Może to być ilość odczytów z bazy danych na sekundę, lub też liczba jednoczesnych użytkowników grających na danym serwerze w jedną grę komputerową.

W momencie, w którym określmy jakie są parametry określające obciążenie systemu, można zacząć zastanawiać się, jak system zareaguje w momencie gdy obciążenie zwiększy się dwukrotnie, lub też czterokrotnie.

1.3.2 Wydajność

Wydajność jest bezpośrednio połączona z obciążeniem. Można ją określić jako adekwatna odpowiedź systemu na dane obciążenie przy danej dostępności zasobów systemowych. Również tak jak obciążenie, mierzenie wydajności zależy od przeznaczenia systemu. Wydajność może być zatem mierzona poprzez czas odpowiedzi systemu na wysłane zapytanie lub też ilość rekordów przetworzonych w ciągu sekundy, czy też ilość zapisów do bazy danych w ciągu sekundy.

Do mierzenia wydajności bardzo przydatne są podstawowe narzędzia statystyki takie jak średnia, czy też mediana. Również inną, często stosowaną metryką są percentile, czyli wartości wydzielające te części próbek, których wartości były niższe od wartości danego percentile'a.

Percentile są przydatne przy obliczaniu jaka jest typowa odpowiedź systemu na przykład w 95% przypadków. Takie narzędzia pomagają w podjęciu decyzji o tym, czy wydajność należy zwiększyć i jak przełoży się to na wyniki biorąc pod uwagę koszty takiej zmiany i uzyskane z niej zyski.

1.3.3 Skalowanie

Sam proces skalowania systemu można określić jako zachowanie tej samej wydajności przy zwiększającym się obciążeniu [9]. W przypadku skalowania wyróżnia się często skalowanie *wertykalne* i skalowanie *horizontalne*.

Skalowanie wertykalne można określić jako zwiększanie możliwości hardware'owych maszyny obsługującej dany system. Można powiedzieć, że zamiast komputera, który może ob-

służyć 1000 zapytań na sekundę udostępniamy komputer, który obsługuje 2000 zapytań na sekundę.

Skalowanie horyzontalne można określić jako dodawanie coraz to większej ilości mniejszych maszyn do naszego systemu. W przypadku już użytego przykładu, do komputera, który obsługuje 1000 zapytań na sekundę dokładamy drugi komputer, który też może obsługiwać 1000 zapytań na sekundę.

Skalowanie może odbywać się automatycznie, w takim wypadku mówimy, że dany system jest elastyczny, lub też manualnie, w tym wypadku mówimy o systemach skalowalnych manualnie. Systemy elastyczne są szybsze, ale za to bardziej skomplikowane niż systemy manualne.

Zazwyczaj droga skalowania najpierw odbywa się w kierunku wertykalnym, a w momencie, w którym jest to niewystarczające, zaczyna odbywać się w kierunku horyzontalnym.

Nie ma jednego typu skalowej architektury”, wszystko zależy od przeznaczenia danego systemu.

1.4 Łatwość w utrzymaniu

Ostatnim problemem jest łatwość w utrzymaniu. Powszechnie wiadomo, że największe koszty jakie niesie ze sobą tworzenie oprogramowania są ponoszone w ostatniej jego fazie, czyli w utrzymaniu [9].

Na utrzymanie składa się łatanie dziur w systemie, dbanie o to, by system działał, reagowanie na awarie, dodawanie nowych funkcjonalności.

W celu zachowania systemu w dobrym stanie wyróżnia się trzy cele projektowe, które sprawiają, że system pozostaje łatwym w utrzymaniu.

- Operacyjność
- Prostota
- Ewolucyjność

1.4.1 Operacyjność

Operacyjność może być określona jako zdolność systemu do ułatwiania powtarzalnych czynności przy jego utrzymaniu [9]. Dzięki temu operatorzy systemu mogą skupić się na innych, bardziej wartościowych czynnościach związanych z rozwojem systemu.

Niektórymi z tych cech, które zwiększą operacyjność systemu są:

- Dobry monitoring systemu.
- Automatyzacja i integracja ze standardowymi narzędziami.
- Możliwość wyłączenia części systemu w celu dokonania jego aktualizacji.
- Dobra dokumentacja.

- Dobry zestaw zachowań domyślnych systemu, jak i łatwość nadpisania wartości domyślnych.
- Możliwość samonaprawy.
- Przewidywalne zachowanie.

1.4.2 Prostota

Prostota systemu oznacza łatwość w pojęciu tego jak system działa i możliwości orientowania się w wewnętrznych jego strukturach [9]. Często systemy w początkowej fazie rozwoju charakteryzują się wysoką prostotą, by z czasem stawać się coraz bardziej złożone.

Złożoność systemu może być powiązana z samym problemem, który system ma rozwiązać, jednakże nie zawsze tak jest. Czasem w systemie istnieje złożoność nadmiarowa, czy też niepotrzebna, która tylko przysparza więcej problemów podczas jego utrzymania.

Niektórymi z powodów zwiększonej złożoności, a tym samym zmniejszonej prostoty systemu są:

- Duża liczba obsługiwanych stanów systemu.
- Bezpośrednie i ścisłe zależności między modułami systemu.
- Splątane zależności międzymodułowe.
- Niekonsekwentne nazewnictwo.

Najczęszym remedium na usuwanie niepotrzebnej zależności jest stosowanie odpowiednich abstrakcji w systemie.

1.4.3 Ewolucyjność

Ewolucyjność to zdolność systemu do zmiany. Wraz z rozwojem systemu często zmieniają się wymagania odnośnie systemu. Najczęszym i najbardziej powszechnym sposobem na zachowanie ewolucyjności systemu jest stosowanie technik *Agile*, do których należą między innymi:

- Test Driven Development
- Refactoring
- Continuous Integration

Zdolność systemu do ewoluowania jest ściśle powiązana z jego prostotą.

1.5 Zastosowanie technik Big Data

Po określeniu problemów związanych z przypadkiem użycia przytoczonym na początku tego rozdziału, można się zastanowić, jakie rozwiązanie jest proponowane przez techniki i sposób myślenia oparty o technologie i narzędzia Big Data [12].

W systemie z przykładu mieliśmy do czynienia z problemem trudności w utrzymaniu i brakiem skalowalności w momencie, w którym system zyskiwał coraz większą popularność, co w ostateczności prowadziło do utraty niezawodności, gdy system został narażony na błąd ludzki.

Po pierwsze w technologiach Big Data z góry przyjmuje się założenie, że będą one używane w systemach rozproszonych. Zatem radzenie sobie z łatwością w utrzymaniu w systemie rozproszonym, czyli łatwo skalowalnym, we wcześniej opisywanym przypadku stają się proste do rozwiązania. Narzędzia Big Data są wyposażone w szereg narzędzi ułatwiających skalowanie i odpowiednie rozłożenie obciążenia.

Inną zasadą działania, często zakładaną w systemach Big Data jest niemutowalność zbioru danych. W momencie, w którym rejestrowane są poszczególne wejścia na stronę zależność RAW (Read After Write), która obecna jest w przypadku modyfikowania jednego licznika w danym wierszu tabeli, znika i jest zastąpiona poprzez operację dopisywania kolejnych wierszy reprezentujących poszczególne wejścia na daną stronę. Dzięki temu uzyskuje się też większą odporność na błędy, gdyż jedyne, co należy zmodyfikować w przypadku błędu programowego, czy też błędu ludzkiego, to nowe dane i nowe wiersze, stare pozostaną nietknięte.

W tym wypadku liczba odwiedzin danej strony jest sumą pojedyńczych wierszy, gdzie każdy reprezentuje pojedyńcze wejście na daną stronę.

Jednakże technologie Big Data to nie tylko zalety, ale także wady. Dobrym przykładem ograniczeń systemów Big Data lub też w ogóle systemów rozproszonych jest tak zwane twierdzenie CAP [2]. Zgodnie z tym twierdzeniem system rozproszony, czyli taki, który posiada warstwę sieciową, w momencie, w którym następuje przerwa w połączeniu między poszczególnymi węzłami systemu, musi wybrać pomiędzy spójnością a dostępnością.

Jeżeli zatem, w systemie rozproszonym Big Data skupimy się na wysokiej dostępności, dzięki horyzontalnej skalowalności, musimy pogodzić się z utratą spójności. W związku z tym, w systemach Big Data nie mówimy o spójności, ale o tak zwanej *ostatecznej spójności*, czyli spójności, którą system osiągnie po bliżej nieokreślonym czasie.

Nie istnieje jedna architektura Big Data, gdyż każde rozwiązanie jakie się stosuje, musi być dostosowane do rozwiązywanego problemu, jednakże twórcy systemów opartych o technologie Big Data wypracowali ogólne modele architektury, które można zastosować w rozwiązywaniu typowych problemów związanych z przetwarzaniem dużych zbiorów danych.

Rozdział 2

Modele architektury Big Data

Nie ma jednej określonej architektury, która rozwiązywałaby wszystkie problemy. Każde zagadnienie, które staramy się rozwiązać przy pomocy technologii przetwarzających duże zbiory danych jest inne. Nawet w przypadku zagadnień, które z goła wydają się podobne, system do ich obsługi, z uwagi na szczegóły, może być zupełnie inny.

Zupełnie inny będzie system służący do obsługi 4 milionów zapytań dziennie po 1 MB każde, a zupełnie inny do obsługi 400 zapytań dziennie po 10 GB każde, pomimo że ilość przesyłanych danych w obu wypadkach jest taka sama.

W tym rozdziale przedstawione zostaną najważniejsze typy architektury, z którymi można się spotkać przy rozwiązywaniu zadań związanych z Big Data, nie są to dokładne modele, a jedynie szkice, na których można się oprzeć przy projektowaniu systemu przetwarzania.

2.1 Podstawowe rodzaje przetwarzania

Zanim przystąpimy do zaprezentowania typów architektury Big Data należy określić dwa podstawowe sposoby przetwarzania danych, które są w nich używane:

- Przetwarzanie wsadowe
- Przetwarzanie strumieniowe

Przetwarzanie wsadowe charakteryzuje się tym, że definiowane jest zadanie, którego celem jest przetworzenie jakiegoś ograniczonego zbioru danych. Zadanie ma swój początek i koniec. Często zadanie wsadowe charakteryzuje się dość długim oczekiwaniem na rezultat, szczególnie wtedy, gdy zbiór danych, który należy przetworzyć, jest bardzo duży.

Przetwarzanie strumieniowe to zadanie, które zostaje rozpoczęte i nie oczekuje się jego zakończenia. Działa ono nieustannie przetwarzając napływające dane. Stan zadania w przetwarzaniu strumieniowym jest przechowywany w pamięci komputera. Wyniki obliczeń są dostępne z minimalnym opóźnieniem, gdyż dane analizowane są na bieżąco.

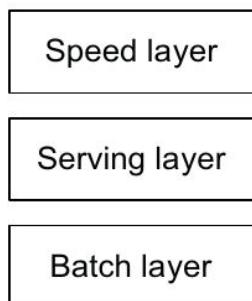
Te dwa podstawowe typy przetwarzania stanowią bazę, na podstawie której będzie zbudowana każda architektura Big Data przedstawiana w tym rozdziale.

2.2 Architektura Lambda

Pierwszą i najlepiej opisaną architekturą Big Data z jaką się możemy spotkać obecnie, jest architektura Lambda. Została ona zaproponowana po raz pierwszy przez Nathana Marza w jego artykule [11], a później dokładniej opisana w książce [12].

Według tego modelu system rozproszonego przetwarzania dużych zbiorów danych składa się z trzech warstw, w których każda wyższa warstwa, bzuje na niższej. Te trzy warstwy to:

- Warstwa szybka (ang. *Speed layer*)
- Warstwa obsługująca (ang. *Serving layer*)
- Warstwa wsadowa (ang. *Batch layer*)



Rysunek 2.1: Warstwy architektury Lambda [12].

Każda z warstw pełni określoną funkcję w systemie i dopiero wszystkie trzy złożone razem spełniają zadanie, dzięki któremu uzyskujemy ostateczną spójność danych pomimo korzystania z rozprozonego systemu przetwarzania danych.

2.2.1 Warstwa wsadowa

Warstwa wsadowa ma dwa podstawowe zadania do wykonania. Pierwszym z nich jest przechowywanie niezmennego zbioru wszystkich danych, które są do niej wysyłane. Drugim jest możliwość obliczenia wartości dowolnej funkcji na dostępnym już zbiorze danych.

Za część odpowiedzialną za przechowywanie dowolnie dużego zbioru danych w systemie rozproszonym najczęściej będzie odpowiadać rozproszony klaster HDFS, natomiast za przetwarzanie tego zbioru danych może odpowiadać np. algorytm MapReduce. Obie te technologie zostaną omówione w dalszej części pracy.

Warstwa wsadowa działa na zasadzie powtarzających się kroków pętli, czyli przetwarza chwilowy obraz powiększającego się zbioru danych, tworząc chwilowe „widoki wsadowe”. Wyniki obliczeń opartych o te widoki wsadowe będą następnie używane do obliczenia dokładnej wartości zapytania.

Ponieważ przetwarzanie wsadowe zajmuje sporo czasu, dlatego ta warstwa jest niewystarczająca by w momencie zapytania otrzymać aktualne wyniki. Z tego też powodu konieczne jest wprowadzenie dwóch kolejnych warstw.

2.2.2 Warstwa obsługująca

Warstwa obsługująca odpowiada za przechowywanie wyników obliczeń dla chwilowych widoków powstających w warstwie przetwarzania wsadowego i udostępnianie operacji odczytu dla tych danych.

Najczęściej jest to wyspecjalizowana rozproszona baza danych. W momencie, w którym nowe widoki zostają dostarczone przez warstwę przetwarzania wsadowego, warstwa obsługująca podmienia aktualnie używane widoki na nowsze.

Warstwa obsługująca nie jest natomiast odpowiedzialna za zapisywanie czegokolwiek do bazy danych, służy tylko do prezentowania i odczytywania widoków stworzonych przez warstwę przetwarzania wsadowego.

Ta warstwa jest najczęściej bardzo prosta.

2.2.3 Warstwa szybka

Ostatnią i najbardziej skomplikowaną warstwą, jest warstwa szybka. Należy sobie uzmysolić, że przetwarzanie wsadowe całego zbioru danych trwa. Jeżeli zależy nam na otrzymywaniu wyników, które są najbardziej zbliżone do rzeczywistości, a takie są najczęściej najbardziej pożąданie, należy znaleźć sposób, na uaktualnienie wyników, obliczonych przez warstwę wsadową i zaprezentowanych przez warstwę obsługującą o te dane, które nadeszły w czasie obliczania wyniku przez te warstwy.

Za to odpowiedzialna jest warstwa szybka. Przetwarza ona dane w strumieniu danych tworząc chwilowe widoki najbardziej aktualnego zbioru danych oraz odpytuje wcześniej wymienione warstwy o widoki obliczone wcześniej, następnie wylicza ona wynik końcowy scalając oba wyniki w jedną całość.

Jest ona najczęściej obsługiwana przez bazę danych umożliwiającą losowy zapis jak i losowy odczyt danych, z uwagi na to jest ona znacznie bardziej skomplikowana od pozostałych dwóch warstw.

Cechą szczególną tej warstwy jest to, że w momencie w którym dane przetworzone tylko w części przetwarzania strumieniowego, dotrą już do systemu przechowywania danych i zostaną zapisane - zostaną dla nich obliczone widoki wsadowe, które ostatecznie, „nadpiszą” wyniki funkcji obliczone dla tych danych w warstwie szybkiej.

2.2.4 Wady i zalety architektury Lambda

Ogromną zaletą architektury Lambda jest izolacja złożoności. Dwie pierwsze warstwy są bardzo proste. Zadania których oczekujemy od tej warstwy są już dobrze poznane i zaimplementowane w popularnych frameworkach opensource, takich jak Hadoop. Jednocześnie obie warstwy spełniają już prawie wszystkie wymagania odnośnie dobrze zbudowanego systemu rozproszonego do przetwarzania dużych zbiorów danych. Są skalowalne, niezawodne i posiadają elastyczność umożliwiającą obliczenie wartości dowolnej funkcji na przetwarzanych danych.

Najbardziej złożona część architektury Lambda jest wydzielona w warstwie szybkiej, która scalą wyniki i prezentuje ostateczny wynik obliczeń. Dzięki takiej dualnej strukturze przy użyciu architektury Lambda otrzymujemy zarówno skalowalność i niezawodność jak i spójność.

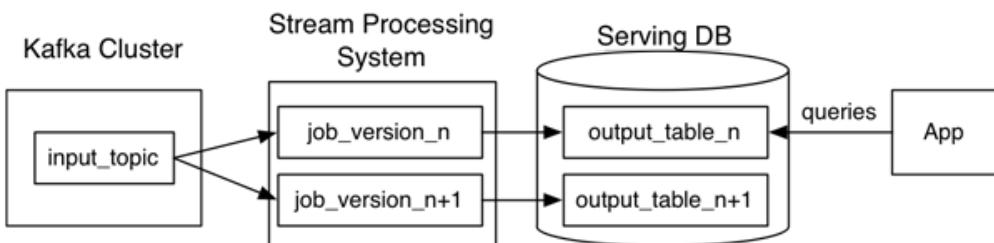
Największą wadą architektury Lambda jest jej duża złożoność jako całości. Konieczność obsługiwanego co najmniej trzech różnych warstw, z których każda ma inne zadanie i dodatkowo każda jest osobnym systemem samym w sobie, jest dość skomplikowane i może przysparzać sporo trudności. Kolejną wadą jest konieczność implementowania logiki obliczania wartości pewnej funkcji w dwóch różnych systemach, zarówno warstwa szybka jak i warstwa wsadowa muszą obliczyć ten sam widok dla danych, które przetwarzają, są to jednak dwa różne systemy i daną funkcjonalność zawsze trzeba zaimplementować dwa razy. Te negatywne cechy skłoniły do poszukiwań nowych rozwiązań architektonicznych i przyczyniły się do powstania kolejnego typu architektury.

2.3 Architektura Kappa

Architektura Kappa została po raz pierwszy zaprezentowana w artykule przez Jay'a Kreps'a [10].

Można powiedzieć, że jest ona uproszczeniem architektury Lambda. Architektura Kappa oparta jest przede wszystkim na przetwarzaniu strumieniowym. Dane nie są zapisywane do rozproszonego systemu przechowywania danych takich jak HDFS (choć mogą być), a są przechowywane jako cały czas dostępne dane strumieniowe.

Strumienie w tej architekturze mogą być bardzo pojemne, rozciągające się na petabajty danych. Podstawowy schemat takiej architektury przedstawia się następująco:



Rysunek 2.2: Architektura Kappa [10].

Jak widać na tym schemacie jedynym systemem przetwarzającym dane jest system przetwarzania strumieniowego. Same dane są przechowywane jako strumień w klastrze Kafki. Kafka zostanie dokładnie opisana w późniejszej części pracy. Można jednak wspomnieć, że jest to rozproszony system przesyłania komunikatów charakteryzujący się między innymi takim parametrem jak *log.retention*, czyli czasem, po którym dana wiadomość jest usuwana z logu Kafki.

W architekturze Kappa parametr *log.retention* posiada wartość tak dużą, że klaszter Kafki przechowuje wszystkie interesujące dane z punktu widzenia systemu przetwarzania, a więc całą historię potrzebną do obliczenia poprawnego wyniku żądanych funkcji. Dzięki temu, w momencie w którym istnieje konieczność obliczenia nowej funkcji na całym zbiorze danych

(jak na przykład liczba wejść od początku istnienia jakiejś strony www), lub zaktualizowania starej, tworzone jest nowe zadanie, które ma przetworzyć cały strumień danych od początku.

Można zadać zatem pytanie, czym różni się to od warstwy wsadowej architektury lambda? Przetwarzanie strumieniowe jest zadaniem oczekującym na nowe dane, w przeciwieństwie do zadań definiowanych w systemie wsadowym, które zaczynają przetwarzanie i kończą przetwarzanie. W systemie przetwarzania wsadowego, by policzyć nową wartość odwiedzin na stronie, należy przetworzyć cały zbiór danych jeszcze raz. W systemie przetwarzania strumieniowego wynik przetwarzania jest cały czas dostępny i jakby zawieszony w oczekiwaniu na nowe dane, które są tylko dopisywane do logu rozproszonego systemu strumieniowego przetwarzania.

Zatem obliczenie tej samej funkcji na nowej porcji danych jest wynikiem przetworzenia tylko tych nowych danych i uzyskanego już wcześniej wyniku funkcji dla wcześniej przetworzonych danych w przetwarzaniu strumieniowym.

Jedynie zadania, których wynik wymaga oczekiwania w przypadku architektury Kappa, to nowo tworzone zadania, które muszę przetworzyć cały strumień danych od początku, jednakże ponieważ nie trzeba się w tym wypadku martwić o to, że jakieś dane zostaną pominięte w przypadku przetwarzania tych danych, gdyż wszystkie zostaną ostatecznie dopisane do jednego strumienia, nie niesie to ze sobą żadnych negatywnych długotrwałych konsekwencji.

Architektura Kappa ma też tę zaletę, że można w niej uruchomić nowszą wersję tego samego zadania i przełączyć się w momencie, w którym nadrobi ono wyniki i zrówna się ze stanem zadania aktualnie działającego. W takim wypadku wystarczy jedynie przełączyć się z jednej wersji zadania na drugą by uaktualnić wyniki.

Oczywiście jak już zostało wspomniane, nie wyklucza to ostatecznego przechowywania danych w systemie HDFS, które mogą być później wykorzystane do dalszego przetwarzania i na przykład algorytmów uczenia maszynowego.

2.3.1 Wady i zalety architektury Kappa

Z pewnością największą zaletą architektury Kappa jest łatwość jej obsługi. Wszystko opiera się na przetwarzaniu strumieniowym strumienia danych z interesującego nas okresu, który może być bardzo długi. W momencie w którym chcemy zdefiniować nową wersję zadania, po prostu je uruchamiamy i czekamy, aż zaktualizuje się ono do stanu zadania, które jest aktualnie w użyciu.

Również ogólną zaletą architektury Kappa, jak i samego przetwarzania strumieniowego, jest szybki czas odpowiedzi, który jest niemal natychmiastowy. Zadanie działa i jego wynik jest nieustannie dostępny w pamięci maszyny.

To podejście do rozwiązywania zadań to duże uproszczenie i ułatwienie, jednakże nie jest ono bez wad.

Największa wada architektury Kappa tkwi w samej specyfice przetwarzania strumieniowego. Przetwarzanie strumieniowe nie radzi sobie dobrze ze wszystkimi przypadkami użycia. Na przykład, w momencie, w którym zbieramy datowane dane z pojazdów autonomicznych i okazuje się, że są one dostarczone nie w takiej kolejności w jakiej zostały wygenerowane, w skutek opóźnień sieciowych [3], w takim wypadku w przetwarzaniu strumieniowym, pewne dane

musiałyby zostać pominięte, gdyż pojawiły się za późno i okienko przetwarzania już zdążyło przejść do późniejszego czasowo okresu.

W przypadku przetwarzania wsadowego w architekturze Lambda wszystkie dane możemy posortować, co nie jest takie łatwe w przypadku przetwarzania strumieniowego.

Kolejnym ograniczeniem jest konieczność przechowywania dużej liczby stanów w przetwarzaniu strumieniowym. Przetwarzanie strumieniowe opiera się na chwilowym stanie przetrzymywany w wykonywanym zadaniu, w momencie, w którym to zadanie ulegnie awarii, dany stan zostaje utracony.

Następnym wyzwaniem jest konieczność porównywania wyników uzyskiwanych w przetwarzaniu strumieniowym z innymi danymi przechowywanymi w zewnętrznych systemach, jak na przykład join pomiędzy danymi obliczonymi w jakiejś funkcji dla, założmy, jakichś pojazdów autonomicznych o określonym identyfikatorze i tabelą identyfikującą te właśnie konkretne pojazdy autonomiczne.

2.4 Podsumowanie

W tym rozdziale zostały przedstawione dwie podstawowe architektury w przetwarzaniu dużych zbiorów danych. Ostateczny system, który zostanie zbudowany powinien wziąć pod uwagę oba podejścia tutaj zaprezentowane i przedstawić rozwiązanie, które najlepiej będzie spełniało stawiane przed nim zadania.

Rozdział 3

Narzędzia Big Data

W przeciągu wielu lat rozwoju Big Data powstało wiele projektów opensource wspomagających i bardzo ułatwiających tworzenie rozproszonych systemów służących do przetwarzania dużych zbiorów danych. Sam rozwój oprogramowania w tej dziedzinie rozpoczął się od powstania Hadoop'a i całego szeregu narzędzi wspomagających. W tym rozdziale zostaną przedstawione najważniejsze i jedne z najnowszych technologii Big Data, za pomocą których później zostanie zrealizowany projekt, który jest tematem tej pracy

3.1 Hadoop

Hadoop to rozproszony system składowania i przetwarzania danych, pierwszy, który zapoczątkował trend Big Data. Na początku projekt ten nosił nazwę Nutch, dopiero później został przemianowany na Hadoop [7].

Hadoop składa się z dwóch podstawowych komponentów: HDFS i YARN, które razem umożliwiają wykonywanie zadań w modelu wsadowym: MapReduce [15].

Na ekosystem Hadoop'a, poza dwoma podstawowymi komponentami, składa się szereg narzędzi takich jak: Hive czy Pig. Nie będą one jednak szerzej omawiane w tej pracy.

3.1.1 Model Map Reduce

Model MapReduce to wsadowy model przetwarzania danych. Sposób przetwarzania w tym modelu składa się z dwóch podstawowych części: mapowania i redukcji.

Mapowanie polega na obliczeniu arbitralnie zdefiniowanej funkcji na małej - jednostkowej porcji danych wejściowych, najczęściej jednym rekordzie.

Redukcja polega na scalaniu wyników pojedyńczych mapowań. Wyniki operacji mapowania są zapisywane jako pliki tymczasowe na dysku twardym.

W modelu MapReduce programista jest odpowiedzialny za zdefiniowanie obu funkcji: mapującej i redukującej. Implementacja tych funkcji odbywa się w Javie.

MapReduce był pierwszym modelem przetwarzania danych wsadowych opartym o technologie Hadoop'a, czyli HDFS i YARN, obecnie jest więcej bardziej zaawansowanych technologii, które charakteryzują się znacznie lepszą wydajnością, niż model MapReduce.

3.1.2 HDFS - Hadoop Distributed File System

HDFS to rozproszony system składowania danych. Opiera się na założeniu jednego zapisu i wielu odczytów (ang. *write once, read many*).

HDFS może być wdrożony na standardowym sprzęcie komputerowym opierającym się na zwykłych dyskach twardych. Charakteryzuje się możliwością zapisu bardzo dużych plików (nawet terabajtowych) jak i dużej ilości mniejszych plików oraz umożliwieniu ich przetwarzania z bardzo dużą efektywnością. Dostęp do danych w systemie HDFS jest realizowany z bardzo małym opóźnieniem.

Sam system HDFS składa się z węzłów nazw i węzłów danych. Węzły nazw są odpowiedzialne za przechowywanie informacji odnośnie tego, gdzie znajdują się poszczególne pliki w systemie i jaka jest hierarchia plików. Węzły danych odpowiadają za same przechowywanie plików i wykonywanie na nich zadań zlecanych w systemie YARN.

Węzły nazw pracują wobec węzłów danych w systemie master - slave. Jeżeli węzeł nazw zostanie usunięty, czy ulegnie awarii, to nie ma możliwości dalszego odczytu plików, dlatego bardzo często tworzy się drugi, zapasowy węzeł nazw, który replikuje podstawowy węzeł nazw. Same dane w systemie HDFS także są replikowane, najczęściej w 3 kopiiach, choć te parametry, jak i wiele innych, są oczywiście konfigurowalne.

3.1.3 YARN - Yet Another Resource Negotiator

YARN to system zarządzania zasobami w Hadoopie. Składa się on z menedżera zasobów i menedżerów węzłów. YARN jest odpowiedzialny za uruchamianie na poszczególnych węzłach klastra zadań zdefiniowanych przez użytkownika. Dzięki temu, że jest to system rozproszony, każde zadanie, może być przetworzone na wszystkich węzłach jednocześnie, co znacznie skracą czas obliczeń.

YARN jest na tyle uniwersalny, że korzystają z niego również inne, późniejsze technologie chcąc zintegrować się z systemem Hadoop. YARN prezentuje uniwersalne API za pomocą którego inne aplikacje mogą zgłaszać do niego zadania.

YARN jest też odpowiedzialny za przydział zasobów dla danego zadania w zależności od konfiguracji zarówno samego klastra YARN jak i konfiguracji samego zadania.

3.2 Spark

Spark jest powszechnie uznawany za następcę modelu MapReduce. Jest to oprogramowanie opensource napisane w języku Scala, które integruje się z systemem Hadoop, ale nie tylko. Został stworzony na Uniwersytecie w Berkeley w 2009 roku [3].

Spark to silnik przetwarzania składający się z wielu bibliotek. Jego podstawową różnicą w porównaniu do systemu MapReduce jest to, że operacje są przechowywane w całości w pamięci. Dzięki temu Spark potrafi być nawet 100 razy szybszy niż model MapReduce, który poszczególne wyniki kroków mapowania i redukcji przechowuje na dyskach [8].

Spark integruje w sobie wiele funkcjonalności, które były rozproszone w systemie Hadoop, a także wprowadza wiele usprawnień i możliwości integrowania się z innym oprogramowaniem.

Jego najważniejsze moduły w najnowszej wersji 3.0 to:

- Przetwarzanie w systemie RDD
- Przetwarzanie w systemie SQL, Dataframe
- SparkStreaming - biblioteka do przetwarzania strumieniowego
- MLlib - biblioteka do uczenia maszynowego
- GraphX - biblioteka do operacji na grafach
- Biblioteka do uczenia głębokiego
- DeltaLake - moduł wprowadzający transakcje ACID do Spark'a

Ogromną zaletą środowiska Spark jest możliwość korzystania ze wszystkich wymienionych narzędzi w jednym miejscu, bez martwienia się o „rozproszony” charakter operacji. Programy pisane w Spark'u wyglądają podobnie do programów pisanych dla systemów nierozerproszonych. Silnik Spark'a troszczy się o to, by operacje były odpowiednio rozdzielane i wysyłane do węzłów w celu przetwarzania.

Programy dla Spark'a mogą być pisane w wielu różnych językach programowania, takich jak: Scala, Python, Java, R, czy też, dzięki odpowiedniemu modułowi, w SQL.

By obliczyć bardziej skomplikowaną operację silnik Spark'a tworzy acykliczny graf skierowany zadania i dzieli go na odpowiednie etapy, te które nie wymagają dostępu do całego zbioru danych i te, które tego wymagają. Następnie przy pomocy własnego zarządcy wysyła je do poszczególnych węzłów, korzystając przy tym np. z API Hadoop'a, gdzie dane zadania są przetwarzane, a następnie ich wyniki, w razie potrzeby, wysyłane z powrotem do maszyny sterującej (*driver*). Ważne jest to, że wartość wszystkich operacji jest obliczana „leniwie” (ang. *lazy evaluation*), czyli dopiero wtedy, gdy ich wynik jest konieczny do dalszej pracy algorytmu.

3.2.1 Resilient Distributed Dataset

To najbardziej niskopoziomowe API Spark'a. Przedstawia dane jako szereg wierszy, które można zinterpretować przy pomocy dowolnej klasy języka w jakim piszemy dany program (w przypadku Pythona np.: string, int, byte).

Pierwsze wersje Spark'a opierały się przede wszystkim na tym API, obecnie nie jest zalecane by go używać, przede wszystkim ze względu na niskopoziomowość. Wysokopoziomowe API Spark'a jak Dataframe jest wewnętrznie tłumaczone właśnie na RDD.

Podstawową cechą RDD jest to, że są to obiekty niemutowalne. Można je przetwarzać tylko by stworzyć nowy obiekt RDD innego typu, ale nie można ich modyfikować.

Na obiektach RDD można dokonywać takich podstawowych operacji jak: redukcja, zliczanie, maximum, minimum, a także specyficzne operacje mapowania biorące pod uwagę partycję na której znajdują się dane.

RDD mają możliwość *cachingu*, czyli chwilowego zachowania obliczonej wartości lub też permanentnego zachowania wyników *persistency*. Jest to ważne, gdyż biorąc pod uwagę to, że w przypadku Spark'a mamy do czynienia z przetwarzaniem funkcyjnym, poszczególne stany pośrednie obliczeń nie są zapisywane. W momencie wywołania tej samej funkcji po raz drugi, wszystkie obliczenia zostaną przeprowadzone jeszcze raz.

3.2.2 DataFrame i SQL

DataFrame to bardziej wysokopoziomowe API niż RDD. Umożliwia ono mapowanie danych na definiowane przez użytkownika klasy, w wyniku czego otrzymuje on do dyspozycji niemutowalny zbiór wierszy i kolumn z przypisanymi typami do każdej kolumny. Umożliwia to operacje podobne do tych wykonywanych w tradycyjnych bazach danych SQL jak złączenia, widoki, sortowanie po kluczu, grupowanie.

Spark SQL umożliwia również zapisywanie kodu tworzonego zadania wprost w języku SQL, który później tłumaczony jest na język Scala. Również wyniki zapytań w Spark SQL są prezentowane w tabelach.

Dataframe to obecnie zalecane API Spark'a jeśli chodzi o przetwarzanie wsadowe.

3.2.3 Spark Streaming

Spark Streaming to narzędzie w środowisku Spark służące do obsługi strumieni danych. W przeciwieństwie do innych narzędzi do obsługi strumieni danych nie jest to „prawdziwe” przetwarzanie strumieniowe, gdyż opiera się ono w istocie na tak zwanym *microbatching*, czyli przetwarzaniu wsadowym na bardzo małych porcjach danych.

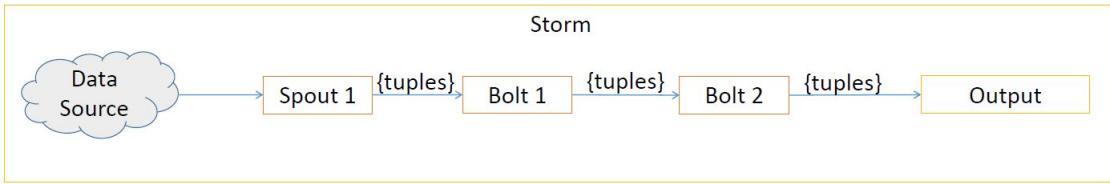
To sprawia, że Spark Streaming posiada nieco dłuższy czas odpowiedzi niż inne narzędzia do przetwarzania strumieniowego.

Zaletami Spark Streaming jest z pewnością to, że jest to bardzo dopracowane narzędzie z bardzo dużym wsparciem i dużą bazą użytkowników.

3.3 Storm

Storm to narzędzie natywnie przeznaczone do przetwarzania strumieniowego [1]. Został wykupiony przez Twittera a następnie udostępniony środowisku opensource. Storm wykorzystuje Zookeeper'a do synchronizacji zadań pomiędzy zarządcą zadań zwanym Nimbus oraz zarządcami na węzłach podległych. Storm działa w technologii master - slave.

Storm tworzy zadania przy pomocy dwóch podstawowych komponentów: strug (*spout*) i belek (*bolt*). Strugi produkują krótki danych, które są następnie wysyłane do belek, które z kolei



Rysunek 3.1: Schemat przedstawiający przepływ danych w Stormie [1].

produkują kolejne krotki. Źródłem krotek dla belek, mogą być inne belki albo strugi. Schemat jest widoczny na rysunku 3.1.

Storm integruje się z różnymi źródłami za pomocą strug, mogą być nimi Kafka, Twitter, czy RabbitMQ. Belki mogą realizować różne zadania takie jak filtrowanie, agregacja i złączenia. Jedna belka może przetwarzać dowolną liczbę strumieni wejściowych. Storm nie wspiera natywnie przetwarzania wiadomości "dokładnie jeden raz" (ang. *exactly once evaluation*), trzeba samemu zadbać o to, by się upewnić, że wiadomość, którą przetwarzamy, nie była już przetworzona wcześniej.

3.4 Flink

Flink jest narzędziem pomyślanym od początku o przetwarzaniu strumieniowym [6]. Jest uznawany powszechnie za następcę Storm'a. Został stworzony na uniwersytecie technicznym w Berlinie w 2015 roku i jest najnowszym z omawianych narzędzi w tej pracy. Podstawową różnicą w porównaniu do Spark'a jest to, że o ile w przypadku Spark'a każde zadanie wewnętrznie jest zadaniem wsadowym, o tyle w przypadku Flink'a każde zadanie jest traktowane jako zadanie przetwarzania strumieniowego. Flink traktuje przetwarzanie wsadowe jako przypadek szczególnego przetwarzania strumieniowego, które ma swój koniec.

Podobnie jak Spark, Flink integruje się z systemem Hadoop, jak i z innymi popularnymi systemami przechowywania danych, jak choćby relacyjne bazy danych, Amazon S3, HBase.

Szybkość przetwarzania danych jest, podobnie jak w Spark'u, nawet do 100 razy szybsza niż w modelu MapReduce.

Flink podobnie jak Spark tworzy swój własny ekosystem z dedykowanymi bibliotekami i modułami odpowiedzialnymi za poszczególne zadania, które chcielibyśmy wykonać na zbiorze danych, a są to:

- DataSet API używany do przetwarzania wsadowego
- DataStream API używany do przetwarzania strumieniowego
- Table - tworzenie zadań w języku SQL
- Gelly - biblioteka grafowa

- FlinkML - biblioteka do uczenia maszynowego

Flink również działa w systemie master - slave. Posiada węzeł odpowiedzialny za zlecanie zadań węzłom podległym, tak zwany *Jobmanager* i *Taskmanager*. Twórcy programu Flink zadbały o bardzo dobre zarządzanie pamięcią. Pomimo tego, że Flink jest uruchamiany w wirtualnej maszynie Javy, zarządzanie pamięcią zostało odpowiednio dostosowane do przetwarzania strumieniowego.

Programy we Flink'u można pisać w Javie, Scali, Pythonie i SQL.

3.5 Kafka

Kafka to rozproszony system przesyłania komunikatów, który spełnia funkcje permanentnego, konfigurowalnego loga. Została stworzona przez LinkedIn by rozwiązać problem przesyłania dużej ilości komunikatów. Dane w Kafce mogą być dopisywane do poszczególnych tematów (ang. *topics*), tworzonych w klastrze Kafki przez producentów, a następnie odczytywane przez konsumentów.

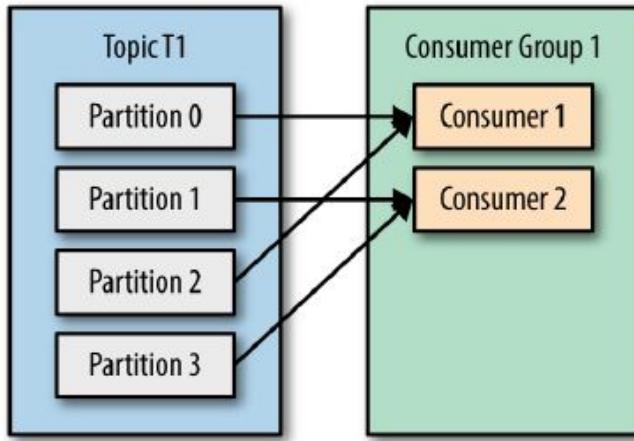
Dane wysyłane do klastra Kafki są nazywane wiadomościami (ang. *messages*). Wiadomości są wysyłane do klastra do odpowiedniego tematu. Tematy są podzielone na partycje, każda partycja może, lecz nie musi, znajdować się na innym serwerze Kafki zwanym brokerem. Każda partycja musi posiadać brokera, który jest odpowiedzialny za zarządzanie daną partycją. Partycje mogą być replikowane. Spośród wszystkich brokerów na których znajdują się repliki danej partycji jeden jest wybierany jako lider. Lider jest właścicielem danej partycji, pozostałe repliki danej partycji mogą być przypisane do innych brokerów, ale nie są oni właścicielami, czyli liderami.

Kafka może obsługiwać zapis przez wielu różnych producentów do jednego tematu jak i odczyt przez wielu różnych konsumentów. Każdy nowy konsument zapisujący się do danego tematu otrzyma wszystkie dane w postaci strumienia danych z Kafki od początku jego istnienia. Można jednak utworzyć grupę konsumentów, wtedy wszyscy uczestnicy grupy partycypują w jednym wspólnym *offsecie*, który jest wskaźnikiem w partycji mówiącym gdzie dana grupa zakończyła odczyt danego tematu. Schemat przedstawiający grupę konsumentów i temat wraz z partycjami jest widoczny na rysunku 3.2.

Kafka zapisuje metadane klastra w Zookeeperze, specjalnego klastra serwerów używanego do synchronizacji systemów rozproszonych. Kafka posiada system retencji, czyli kryteria tego kiedy i w jakich warunkach pozbywać się poszczególnych wiadomości z tematów. Można ustawić w tym wypadku kryteria czasowe jak i ilościowe.

3.5.1 Kafka Streams

Kafka posiada własny moduł odpowiedzialny za przetwarzanie strumieniowe. Jest to moduł, który nie wymaga dodatkowego klastra do działania i natywnie integruje się z Kafką. Umożliwia przetwarzanie strumieniowe danych z tematów Kafki i zapisywanie ich z powrotem do



Rysunek 3.2: Schemat przedstawiający grupę konsumentów [13].

Kafki. Wydajnością nie dorównuje Flink'owi, czy Spark'owi, jednakże jako, że jest to narzędzie wbudowane może być przydatne w sytuacji, gdy nie mamy zbyt dużych wymagań odnośnie framework'a strumieniowania i dodatkowych narzędzi, które mogłyby ono nieść ze sobą.

3.6 RabbitMQ

RabbitMQ to rozproszony system przekazywania komunikatów [14]. W przeciwieństwie do Kafki nie istnieje tutaj możliwość długotrwałego przetrzymywania wiadomości, gdyż są one usuwane z kolejki brokera w momencie gdy zostaną zaakceptowane przez konsumenta. RabbitMQ sam odpowiada za wysyłanie wiadomości do konsumentów, w przeciwieństwie do Kafki, w której to konsumenti odpowiadają za odczytywanie wiadomości z kolejki. RabbitMQ jest prostszym systemem niż Kafka. RabbitMQ nie oferuje możliwości dostarczenia wiadomości dokładnie jeden raz. Najczęstszym zastosowaniem RabbitMQ jest komunikacja między aplikacjami.

3.7 HBase i Apache Phoenix

HBase to kolumnowa baza danych, której system plików jest umieszczony na klastrze HDFS. Baza kolumnowa oznacza, że w przeciwieństwie do zwykłej, relacyjnej bazy danych, dane nie są przechowywane w wierszach, a w kolumnach. Schematyczny sposób zapisu danych w takiej bazie jest przedstawiony na rysunku 3.3.

Jak można zauważyć zapisywana jest ilość zer i jedynek po kolei dla danej wartości w kolumnie, gdzie zero oznacza, że dana wartość w danym wierszu nie występuje, natomiast jeden oznacza, że w danym wierszu dana wartość występuje, wartości ilości zer i jedynek są zapisywane po kolei naprzemiennie.

Baza danych kolumnowa jest bardzo przydatna w przypadku bardzo dużej ilości wierszy i relatywnie małym zbiorze wartości, mogących zostać zapisanych w tych wierszach. W bazie kolumnowej operacje na grupie wartości w danej kolumnie są wykonywane bardzo szybko.

Column values:

product_sk: 69 69 69 69 74 31 31 31 31 29 30 30 31 31 31 68 69 69

Bitmap for each possible value:

product_sk = 29: 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

product_sk = 30: 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

product_sk = 31: 0 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0

product_sk = 68: 0 1 0 0 0 0 0 0

product_sk = 69: 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0

product_sk = 74: 0 0 0 0 1 0

Run-length encoding:

product_sk = 29: 9, 1 (9 zeros, 1 one, rest zeros)

product_sk = 30: 10, 2 (10 zeros, 2 ones, rest zeros)

product_sk = 31: 5, 4, 3, 3 (5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)

product_sk = 68: 15, 1 (15 zeros, 1 one, rest zeros)

product_sk = 69: 0, 4, 12, 2 (0 zeros, 4 ones, 12 zeros, 2 ones)

product_sk = 74: 4, 1 (4 zeros, 1 one, rest zeros)

Rysunek 3.3: Schemat sposób zapisu danych w bazie kolumnowej [9].

Jednakże sama baza danych HBase jest dość trudna w obsłudze, gdyż korzysta jedynie z nie-wygodnego API opartego o pliki XML. By móc korzystać wygodnie z takiej bazy danych należy ją dodatkowo wyposażyć w silnik zapytań SQL, czyli w tym wypadku Apache Phoenix. Język zapytań używany przez Apache Phoenix nie jest do końca zgodny ze standardem SQL, jednakże jest na tyle podobny, że w zupełności wystarcza i zdecydowanie ułatwia operacje wykonywane na tej bazie danych.

3.8 Podsumowanie i wybór narzędzi

By wybrać odpowiednie narzędzia należy najpierw spojrzeć jakie zastosowania i wymagania zostały przewidziane dla naszego systemu zarządzania.

Po pierwsze należy zadbać o to, by wiadomości z pojazdów autonomicznych i mgły obliczeniowej dotarły do miejsca docelowego z jednoczesną możliwością udostępnienia ich silniowi wnioskowania w razie potrzeby. Od razu widać, że należy wybrać system umożliwiający wielokrotny odczyt tego samego strumienia danych różnym odbiorcom, zatem jeśli chodzi o platformę odpowiedzialną za strumieniowanie danych wszystko wskazuje na klaszter Kafka a nie RabbitMQ.

Jednym z zadań jest również przechowywanie całego zbioru danych do dalszego przetwarzania, co samo z siebie narzuca permanentny rozproszony storage taki jak system HDFS.

Ostatnim oczywistym wyborem, którego należy dokonać to wybór środowiska przetwarzają-

nia danych z Kafki. Storm wydaje się być już przestarzałym rozwiązaniem, dodatkowo nie mającym możliwości zapewnienia przetworzenia wiadomości dokładnie jeden raz. Storm nie ma też tak kompleksowych bibliotek jak Flink. Dlatego w przypadku narzędzia do przetwarzania strumieniowego właściwym narzędziem wydaje się być Flink.

W przypadku narzędzia, które ma służyć do przetwarzania wsadowego wybór narzuca się samoistnie. MapReduce jest już bardzo przestarzałym algorytmem i bardzo wolnym. Wymaga bardzo wiele pracy jeśli chodzi o synchronizację zadań pomiędzy poszczególnymi etapami mapowania i redukcji. Spark jest zdecydowanie bardziej wydajny i z ogromnym wsparciem, dlatego wydaje się być właściwym narzędziem do wykonywania zadań wsadowych.

Dodatkowym narzędziem, które okaże się przydatne w projekcie będzie baza danych HBase wraz z silnikiem SQL Apache Phoenix. Jest ona dostosowana do przechowywania i odpytywania dużych zbiorów danych, będzie ona pełniła rolę bazy przechowującej widoki wsadowe, jak w przypadku architektury Lambda i warstwy wsadowej.

3.9 Narzędzia pomocnicze

By móc właściwie korzystać z niektórych, opisywanych wyżej narzędzi, lub też poprawnie zarządzać konfiguracją, konieczne jest skorzystanie z pewnych dodatkowych pomocniczych narzędzi, które w niektórych wypadkach są koniecznością, a w innych preferencją autora tej pracy.

3.9.1 Zookeeper

Pierwszym narzędziem pomocniczym jest Zookeeper. Jest to narzędzie używane przez prawie wszystkie wcześniej wymienione narzędzia. Implementuje ono podstawowe algorytmy konieczne do niezawodnego działania systemów rozproszonych takie jak, wybór lidera w systemie rozproszonym czy też algorytm przesyłania komunikatów w jednym porządku (ang. *total order broadcast*). Nie ma sensu implementować tych algorytmów na nowo (są one dość skomplikowane do zaimplementowania), jeśli zostały one już raz, dobrze zaimplementowane.

3.9.2 Ansible

Kolejnym pomocniczym narzędziem używanym w tej pracy, jest narzędzie do zarządzania konfiguracją, czyli Ansible. Zostało ono wybrane przede wszystkim na prostotę instalacji i w miarę zrozumiałą język konfiguracji i tworzenia skryptów w tym narzędziu. Jest to narzędzie napisane w Pythonie, posiada obszerną dokumentację i w bardzo prosty i szybki sposób można w nim zdefiniować wszystkie podstawowe, jak i bardziej zaawansowane operacje. Jest to też najlepiej znane narzędzie do zarządzania konfiguracją przez autora pracy.

Rozdział 4

Projekt i funkcje systemu

Zanim zostanie przedstawiona implementacja realizowanego systemu należy się przede wszystkim zastanowić nad funkcjami, które system ma realizować oraz nad projektem systemu. Z uwagi na dużą złożoność projektu, liczba funkcji została ograniczona do niezbędnego minimum, takiego, które tworzy rdzeń systemu umożliwiając łatwą rozbudowę o kolejne funkcje i pokazuje ogólną koncepcję działania takiego systemu.

4.1 Wymagania

Ponieważ funkcje realizowane przez system zawsze będą realizowane na dużych zbiorach danych, interfejs wejścia/wyjścia rozpatrywanego systemu musi również być interfejsem potrafiącym przetworzyć tak dużą ilość danych. W związku z tym zastosowanie tradycyjnej bazy relacyjnej jest niewystarczające, co zostało już opisane w analizie przypadku użycia (1.1). Konieczne jest zastosowanie takich narzędzi, które umożliwią właściwy, niezawodny przepływ danych. Jednocześnie zapewniają, że dane nie znikną w przypadku awarii jednego z serwerów, by spełnić te wymagania, stosowane narzędzia muszą zapewniać zdolność replikacji danych.

Kolejnym z wymogów tworzonego systemu jest konieczność katalogowania i obliczania różnych, arbitralnych funkcji na tworzonym zbiorze danych, które mają posłużyć do optymalizacji algorytmów używanych w pojazdach autonomicznych. By spełnić te wymagania istnieje konieczność zastosowania "bazy danych" mogącej przechować bardzo różne zbiory danych, zaczynając od danych z czujników prędkości, a kończąc na obrazach z kamer i lidarów. Tego typu dane noszą nazwę półstrukturyzowanych lub też niestrukturyzowanych. Tworzony system musi mieć możliwość przetworzenia wszystkich tych rodzajów danych. Należy też zaznaczyć, że w kolejnych wersjach pojazdów mogą pojawić się kolejne, nowe źródła danych, w związku z tym tworzony system musi mieć możliwość łatwego dodania nowego źródła dla nowych danych.

Innym aspektem tego zagadnienia jest też efektywne przetwarzanie dużego zbioru danych przez dane narzędzie. Tematem powiązanym z danymi i optymalizacją algorytmów jest wstępne przetwarzanie danych przez system. By móc efektywnie optymalizować tworzone algorytmy należy najpierw dane wstępnie obrobić. Obróbka danych, czyli tak zwany *preprocessing* polega najczęściej na odfiltrowaniu danych odstających lub błędnych, czy też, w przypadku gdy

chcemy nauczyć nasz algorytm lepszej jazdy na przykład nocą, wybraniu tylko tych zapisów z kamer, które dotyczą tylko pewnej pory dnia. To definiuje nam nowe wymaganie dla naszego systemu, którym jest możliwość definiowania dowolnej, arbitralnie filtrującej funkcji dla zbiorów danych, które będą przetwarzane. W przypadku bardzo rozbudowanego systemu, można by tę funkcjonalność *preprocessingu* rozszerzyć już na sam *processing*. Tworzony system prześlabyć być już tylko systemem filtrującym, ale też systemem, który wykonuje pełnię obliczeń wymaganych przez użytkownika na przechowywanych danych.

Kolejnym z wymogów jest czas odpowiedzi. Tworzony system musi mieć możliwość szybkiej odpowiedzi w przypadku, gdy jest to niezbędne, jak w przypadku np. próby poszukiwania objazdu w momencie, gdy droga, którą dany pojazd ma się poruszać, jest zablokowana, np. wskutek niedawnego wypadku.

4.2 Funkcje systemu

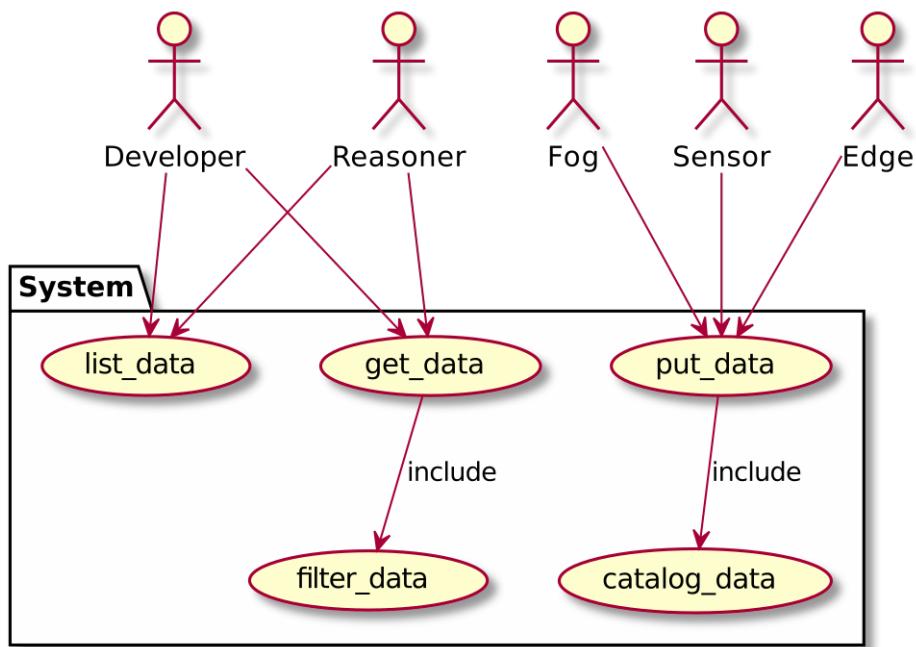
Liczba funkcji systemu, wynikająca z wymagań jest bardzo długa, jednakże z racji ograniczonej ilości czasu, w przypadku rzeczywistego prototypu tworzonego systemu, należy tę liczbę ograniczyć do niezbędnego minimum, które z jednej strony będzie na tyle duże, by pokazać możliwości systemu i na tyle małe, by mogło zostać od początku do końca zaprojektowane i zaimplementowane przez jedną osobę. Jednocześnie taki projekt po zrealizowaniu powinien uwydatniać cały potencjał i możliwości, które tworzony system miałby, gdyby był tworzony jako rozwiązanie produkcyjne.

Bazując na tym, można założyć, że system powinien spełniać dwie podstawowe funkcje. Pierwsza z nich to odbieranie danych ze źródeł danych, następnie katalogowanie ich, oraz wstępne przetwarzanie w celu obliczenia funkcji, które powinny być dostępne dla użytkownika natychmiastowo, jedną z takich funkcji powinna być lista wszystkich przechowywanych danych. Gdyby system nie posiadał tej funkcji uzyskanie szybko informacji o jakichkolwiek specyficznych danych wiązałoby się z przeglądnięciem całego zbioru danych i stworzeniem listy, co, w przypadku bardzo dużej ilości danych, prawdopodobnie zajęłoby bardzo dużo czasu.

Ta część systemu, w której gromadzone są dane i następnie dla nich obliczane są arbitralnie zdefiniowane funkcje, by w kolejnym kroku umieścić je w szybkiej bazie danych, jest wprost związana z architekturą Lambda. Jest to warstwa przetwarzania wsadowego oraz warstwa usług.

Kolejną z funkcji realizowanego systemu jest pobieranie interesujących nas danych przefiltrowanych przez jakieś zapytanie. W wypadku rozpatrywanego problemu mogłaby być to chęć pobrania przez użytkownika danych wszystkich samochodów znajdujących się w danym regionie na świecie, np. w Europie.

Te dwie podstawowe funkcje nie są wystarczające, jednakże stanowią dobrą bazę, na podstawie której można definiować inne ważne funkcje systemu. Diagram przypadków użycia dla tak tworzonego systemu przedstawia się tak jak na rysunku 4.1.



Rysunek 4.1: Diagram przypadków użycia

Po określeniu podstawowych funkcji systemu można przejść do przedstawienia wstępnego projektu rozwiązania.

4.3 Projekt rozwiązania

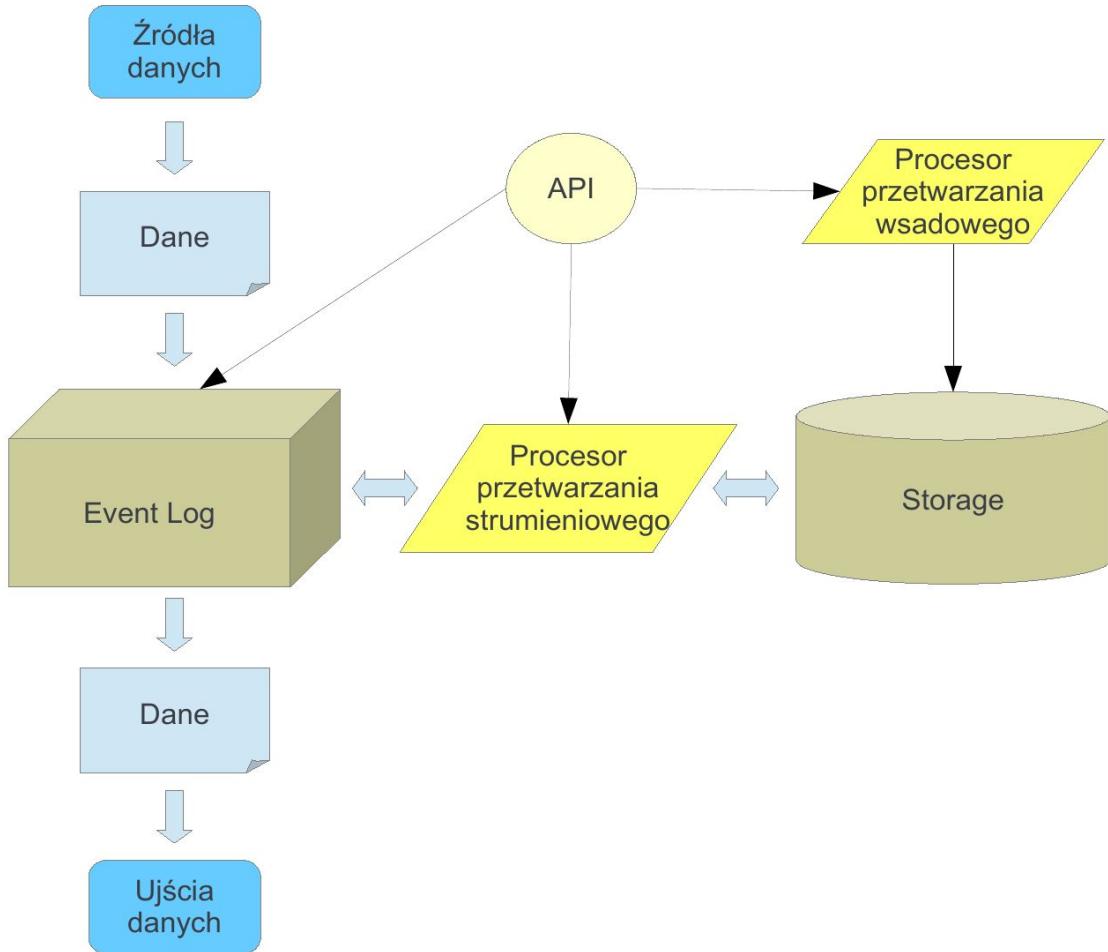
Pierwszym elementem niezbędnym każdego tworzonego systemu przetwarzania, jest interfejs wejścia wyjścia. Rolę takiego interfejsu, spełniającego wcześniej nakreślone wymagania, będzie pełnił klaster Kafki. Posiada on funkcje replikacji danych, oraz bardzo szybki czas odpowiedzi.

Oprócz tego konieczne będzie też użycie systemu przetwarzania strumieniowego dla odpowiedzi, które są krytyczne pod względem czasu. Ta część systemu będzie realizowana przez klaster Flink'a.

System przechowywania danych i systemu przetwarzania wsadowego, istotny z punktu widzenia późniejszego przetwarzania i optymalizacji algorytmów, będzie realizowany przez klaster HDFS i Spark.

Wszystkie te narzędzia będą musiały być zarządzane przez zewnętrzny orkiestrator, tak by ograniczyć liczbę zapytań do różnych API, oraz ujednolicić funkcje realizowane przez system. Stworzone API orkiestratora nie będzie interfejsem wejścia/wyjścia, a jedynie punktem komunikacyjnym i zarządzającym całym systemem. W przypadku zapytania, które będzie polegało na chęci pobrania jakiegoś zbioru danych, orkiestrator uruchomi odpowiedni proces przenoszący dane z systemu przechowywania danych na interfejs wejścia/wyjścia, czyli Event Log realizowany przez Kafkę.

Ogólny schemat rozwiązania jest przedstawiony na rysunku 4.2.

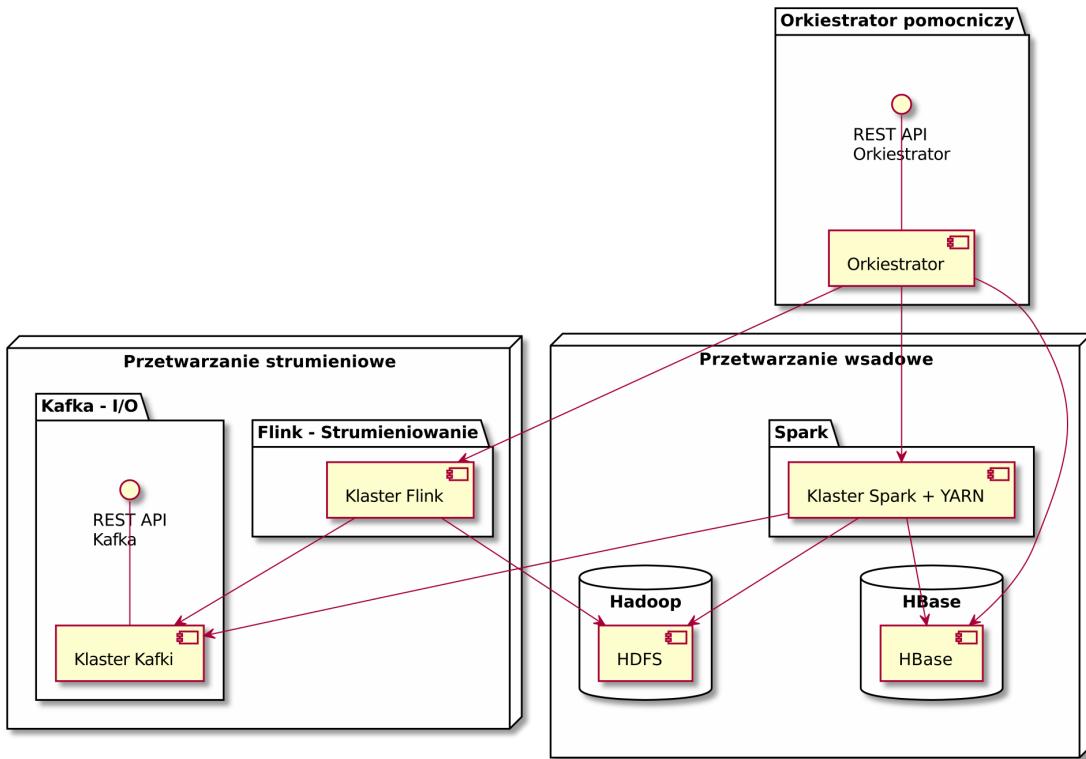


Rysunek 4.2: Ogólny schemat rozwiązania

Poszczególne części systemu w zależności od funkcji byłyby realizowane przez następujące, wcześniej już opisane narzędzia:

- Event Log - Klaster Kafka
- Storage - Klaster HDFS
- Procesor przetwarzania wsadowego (Batch processor) - Klaster Spark + YARN
- Procesor przetwarzania strumieniowego (Streaming processor) - Klaster Flink
- Baza danych do katalogowania (Storage) - Klaster HBase

Dane ze źródeł danych są przekazywane do Event Log'a, czyli do klastra Kafki, gdzie mogą być już wstępnie przetwarzane przez system przetwarzania strumieniowego. Następnie, również za pomocą systemu przetwarzania strumieniowego są przekazywane do systemu przechowywania danych (Storage), czyli do klastra HDFS. Tam system przetwarzania wsadowego będzie mógł przetwarzać już zgromadzone dane i odpowiednio wcześniej zdefiniowane funkcje obliczać i umieszczać w bazie danych HBase, gotowe do wyeksportowania z powrotem do klastra Kafki, czyli interfejsu wejścia/wyjścia.



Rysunek 4.3: Diagram komponentów

Dane zarówno z samego zbioru danych jak i z bazy danych będą wysyłane z powrotem do Kafki również za pomocą procesora strumieniowania danych.

Diagram komponentów z rozbiciem na zdefiniowane narzędzia, wraz z zależnościami jest przedstawiony na rysunku 4.3.

Dodatkowym elementem nieuwzględnionym na diagramie jest klaster Zookeeper'a, który jest pomocniczym klastrem wymaganym do działania pozostałych komponentów.

Jak widać na diagramie komponentów (Rys. 4.3) to orkiestrator jest odpowiedzialny za dostęp zarówno do systemu odpowiedzialnego za przetwarzanie wsadowe jak i do systemu odpowiedzialnego za przetwarzanie strumieniowe. Dzięki temu użytkownik nie ma dostępu do obiektów tworzonych i umożliwiających działanie przez te narzędzia, a jedynie do wysokopoziomowego modelu danych określonego na potrzeby funkcji realizowanych przez system.

Za przesyłanie danych z klastra Kafki do klastra HDFS będzie odpowiedzialny Flink. Za przetwarzanie wsadowe będzie odpowiedzialny Spark, za przechowywanie wyników przetwarzania będzie odpowiedzialna baza danych HBase. Będzie ona również odpowiedzialna za przechowywanie danych operacyjnych orkiestratora, głównie ze względu na konieczność ograniczenia używanych narzędzi w systemie.

Natomiast za przesyłanie danych z powrotem z HDFS do Kafki, oraz z HBase do Kafki może być odpowiedzialny zarówno Spark i Spark Streaming jak i Flink. Decyzja o tym, którego procesora do przetwarzania strumieniowego wybrać w tym wypadku powinna zależeć już od dalszych decyzji podjętych na etapie realizacji infrastruktury.

4.4 Model danych

Sama infrastruktura nie jest kompletnym rozwiązaniem problemu. Tworzy ona dość skomplikowaną kombinację niezależnie działających części. W celu poprawnego i efektywnego wykorzystania wszystkich narzędzi należy stworzyć warstwę abstrakcji, która umożliwi uproszczony i zarazem efektywny dostęp do wszystkich zasobów realizujący zamierzone cele systemu. Konieczne zatem jest stworzenie modelu danych na tyle elastycznego, by umożliwiał użycie systemu w wielu różnych przypadkach, w szczególności zaś by umożliwiał zrealizowanie zarządzania danymi dla pojazdów autonomicznych. Jednocześnie konieczne jest by był on użyteczny dla zwykłego użytkownika i znacznie upraszczał komunikację z systemem.

Dobry model danych powinien też umożliwiać dalszy rozwój, nie powinien być zamknięty na rozszerzenia funkcjonalności. Model danych, w tym wypadku, spełnia rolę abstrakcyjnego interfejsu w przepływie sterowania w programie. Jest podstawową jednostką komunikacji między poszczególnymi komponentami systemu.

W przypadku systemu zarządzania danymi dla pojazdów autonomicznych sprawa wydaje się dość prosta. Podstawową jednostką, która będzie używane w naszym systemie są właśnie „dane” bardzo różnego rodzaju i struktury. Dobrą abstrakcją zatem, będzie coś, co jednoznacznie określa te dane, nadaje im kształt i strukturę. W tej pracy zostało to określone jako „zbiór danych” (ang. *dataset*).

Zbiór danych posiada takie atrybuty jak:

- wejściowy strumień
- zdefiniowane tabele
- strumienie wyjściowe

Strumienie wejściowe jak i wyjściowe są realizowane za pomocą klastra Kafki. Tabele natomiast są realizowane w bazie danych HBase wraz z możliwością przetwarzania zapytań SQL dzięki oprogramowaniu Apache Phoenix.

4.4.1 Strumień wejściowy

Strumień wejściowy to strumień danych zdefiniowany dla danego zbioru danych, który zbiera dane i przekazuje je do miejsca przechowywania danych. W niniejszej pracy, każdy zbiór danych posiada tylko jeden wejściowy strumień danych reprezentowany przez odpowiedni temat w klastrze Kafki. Oczywiście nie oznacza to, że dane mogą być pobierane tylko z jednego źródła. Klaster Kafki ma możliwości by przyjmować dane z wielu różnych źródeł, jednakże ostatecznie te dane są przetwarzane tylko w jednym strumieniu wejściowym z punktu widzenia systemu.

4.4.2 Zdefiniowane tabele

Do każdego zbioru danych może być przyporządkowanych wiele tabel. Tabele tym różnią się od samego zbioru danych, że mogą reprezentować jakieś z góry ustalone funkcje. Np. w przypadku pojazdów autonomicznych, jeśli przekazywane są informacje odnośnie prędkości pojazdu, to w zdefiniowanej tabeli może się znajdować informacja o średniej prędkości danego pojazdu.

Tabele z zasady są zawsze opóźnione wobec zbioru danych jeśli chodzi o aktualność zawartych w nich informacji. Są one tworzone poprzez nieustannie uruchamiane zadanie wsadowe w Spark'u. Oczywiście istnieje możliwość obok zadania wsadowego, zdefiniować inne zadania strumieniowe dla danego zbioru danych tak, by nie było dużego opóźnienia danych zawartych w tabeli wobec danych zawartych w zbiorze danych. Wtedy taką funkcjonalność należałoby realizować poprzez moduł Spark Structured Streaming.

W niniejszej pracy dodano tylko podstawowe możliwości przetwarzania, takie jak wybieranie poszczególnych kolumn ze zbioru danych i umieszczanie ich w odpowiednich tabelach, a także stosowanie takich podstawowych funkcji SQL jak COUNT, AVG itp.

4.4.3 Strumienie wyjściowe

Każdy zbiór danych jak i każda tabela może mieć zdefiniowanych wiele strumieni wyjściowych. W przypadku zbioru danych na strumień wyjściowy mogą być nałożone dodatkowe warunki filtracji, podobnie jak w przypadku definiowania tabel. W przypadku tabel natomiast nie ma takie możliwości. Przyjęto założenie, że sama tabela jest już docelową formą danych i istnieje możliwość jedynie na jej wysłanie do strumienia/strumieni wyjściowych. W przypadku tabel nie istnieje możliwość wysyłania nieustanego, natomiast w przypadku zbioru danych istnieje taka możliwość.

Rozdział 5

Realizacja

Należy mieć na uwadze to, że prezentowane rozwiązania w tej pracy są tylko szkieletem, założkiem rzeczywistych rozwiązań produkcyjnych. Mają one nakreślić podstawowe założenia i ramy rzeczywistego projektu. By stworzyć rzeczywisty system działający w warunkach produkcyjnych należałoby zatrudnić do tego przynajmniej 3 pełne zespoły developersko/testerskie pracujące na pełny etat, które mogłyby ten system rozwijać i poprawiać nieustannie. 2 zajmowałyby się infrastrukturą (której rozwój w tym projekcie zajmował ok. 70% czasu, natomiast 1 mógłby zająć się rozwojem orkiestratora).

5.1 Infrastruktura

W przypadku opisywanego systemu praca infrastrukturalna zajęła czasowo najwięcej. Wszystkie opisywane narzędzia są bardzo skomplikowane w konfiguracji, uruchomienie choćby jednego klastra zazwyczaj wiąże się z szeregiem problemów, które najczęściej rozwiązuje się poprzez przeglądanie logów i szukaniu błędów. Bardzo wiele błędów nie daje właściwego komunikatu. To często dość żmudna praca domyślania się przyczyny awarii danego systemu.

By można było z sukcesem uruchomić wszystkie opisywane narzędzia, za pomocą których realizowany jest dany system, konieczne było skorzystanie z narzędzia do zarządzania konfiguracją, opisywanym już we wcześniejszym rozdziale, t.j. Ansible. Dla narzędzia Ansible został stworzony podstawowy plik z hostami oraz szereg scenariuszy służących do instalacji i aktualizacji danych narzędzi Big Data.

5.1.1 Szczegółowa realizacja rozwiązania

Ostatecznie stworzone zostały następujące klastry:

- Hadoop 3.3.0:

- HDFS

- YARN

- Spark 2.4.8

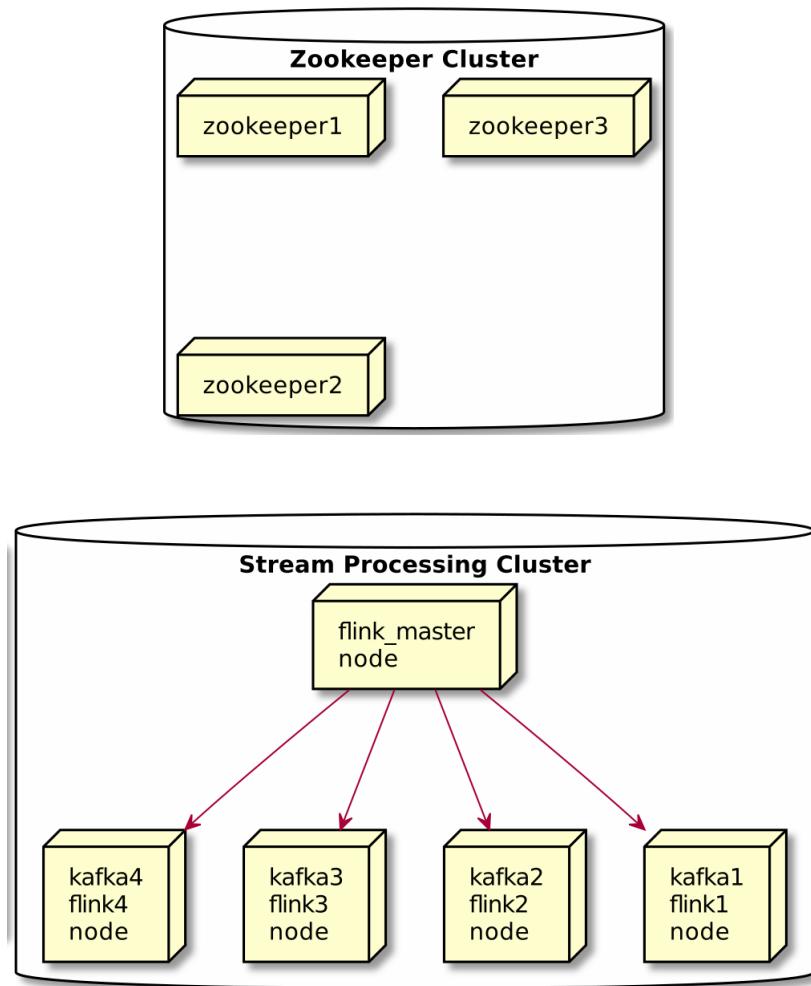
- HBase 2.4.2
- Kafka 2.5.0
- Flink 1.13.0
- Zookeeper 3.6.1

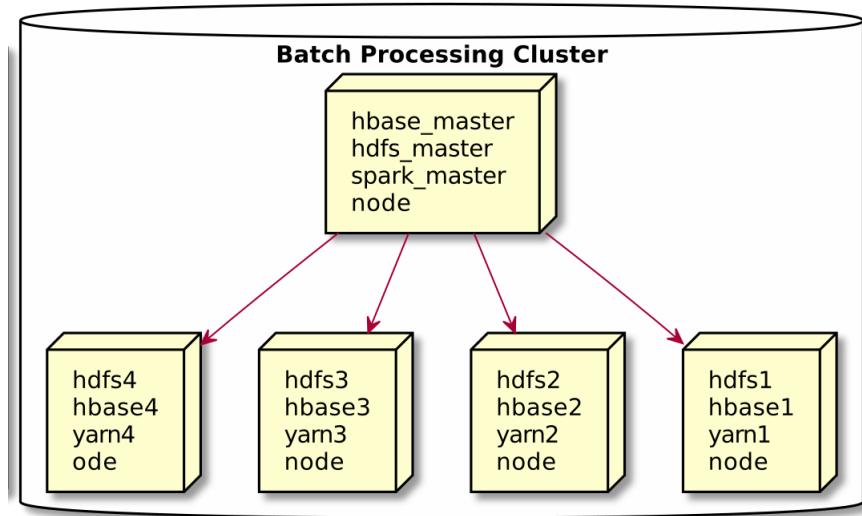
Oprócz powyższych klastrów zostało też uruchomione oprogramowanie pomocnicze:

- Apache Phoenix Queryserver 6.0.0.7.2.9.0-203
- Kafka REST 6.0.0

Węzły klastrów Spark, Hadoop (YARN + HDFS) oraz HBase są współdzielone ze względu na bliską współpracę ze sobą powyższych systemów. Również węzły klastra Flink, Kafka oraz Phoenix Queryserver zostały zrealizowane na jednych maszynach. Zookeeper jako klaster jest współdzielony przez niektóre pozostałe systemy (Flink, HBase, Kafka), ze względu na to został wydzielony jako osobny klaster z osobnymi węzłami.

Tak więc schemat węzłów przedstawiony jest na rysunku 5.1.





Rysunek 5.1: Wykres infrastruktury

Fragment stworzonego pliku z hostami jest widoczny na listingu 5.1.

```
[zookeeper:vars]
ansible_become=true
ansible_become_pass="{{ host_sudo_pass }}"
ansible_port=22

[zookeeper]
zookeeper1 ansible_host=192.168.2.11 myid=1
zookeeper2 ansible_host=192.168.2.12 myid=2
zookeeper3 ansible_host=192.168.2.13 myid=3

[kafka:vars]
ansible_become=true
ansible_become_pass="{{ host_sudo_pass }}"
ansible_port=22

[kafka]
kafka1 ansible_host=192.168.2.101 myid=1
kafka2 ansible_host=192.168.2.102 myid=2
kafka3 ansible_host=192.168.2.103 myid=3
kafka4 ansible_host=192.168.2.104 myid=4

[flink:vars]
ansible_become=true
ansible_become_password="{{ host_sudo_pass }}"
ansible_port=22

[flink]
flink1 ansible_host=192.168.2.121
flink2 ansible_host=192.168.2.101
```

```

flink3 ansible_host=192.168.2.102
flink4 ansible_host=192.168.2.103
flink5 ansible_host=192.168.2.104

```

Listing 5.1: Fragment stworzonego pliku z hostami.

Przykładowy skrypt służący do aktualizacji konfiguracji Kafki na węzłach przedstawiony jest na listingu 5.2.

```

- name: Kafka Update
  hosts: kafka
  vars:
    app_dir: "{{ inventory_dir }}/kafka"
  vars_files:
    - "{{ inventory_dir }}/common/credentials/passwd.yml"
    - "{{ inventory_dir }}/common/variables/common.yml"
    - "{{ app_dir }}/variables/kafka.yml"
  tasks:
    - name: Copy configuration file
      template:
        src: "{{ app_dir }}/config_files/server.properties.j2"
        dest: "{{ KAFKA_HOME }}/config/server.properties"
        owner: kafka
        group: kafka
        mode: '0644'

    - name: Allow firewall ports
      ansible.posix.firewalld:
        port: "{{ item }}"
        permanent: yes
        state: enabled
  loop:
    - 9092/tcp
    - 9000/tcp
    - 9999/tcp

    - name: Restart firewall
      systemd:
        name: firewalld
        state: reloaded

    - name: Copy service file for Kafka
      template:
        src: "{{ app_dir }}/config_files/kafka.service.j2"
        dest: /usr/lib/systemd/system/kafka.service

    - name: Copy service file for CMACK
      when: myid == 1
      template:
        src: "{{ app_dir }}/config_files/cmak.service.j2"

```

```

dest: /usr/lib/systemd/system/cmak.service

- name: Reload deamon
  systemd:
    daemon_reload: yes

- name: Restart kafka.service
  systemd:
    name: kafka
    state: restarted
    enabled: yes

- name: Restart cmak service
  when: ansible_host == CMAK_HOST
  systemd:
    name: cmak
    state: restarted
    enabled: yes

```

Listing 5.2: Skrypt napisany w Ansible służący do aktualizacji konfiguracji Kafki.

Jak widać narzędzie Ansible umożliwia nam korzystanie z dynamicznych zmiennych do tworzenia konfiguracji w zależności od tego, na jakim węźle ta konfiguracja będzie się znajdowała. W przypadku Kafki konieczne jest przypisanie do każdego brokera unikalnego pola "myid". Dzięki Ansible możemy te pola albo stworzyć dynamicznie w pętli, albo użyć zmiennych przypisanych do każdego węzła, jak to jest widoczne na listingu 5.3. Ansible korzysta z narzędzia do szablonowania, które umożliwia dynamiczne uzupełnianie plików konfiguracyjnych zmiennymi wartościami przypisanymi do każdego z węzłów.

```

listeners=PLAINTEXT://{{ ansible_host }}:{{ KAFKA_PORT }}
advertised.listeners=PLAINTEXT://{{ ansible_host }}:{{ KAFKA_PORT }}
broker.id={{ myid }}
min.insync.replicas=2
num.network.threads=3
num.io.threads=8
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
log.dirs={{ KAFKA_HOME }}/log_dir
num.partitions={{ kafka_partitions_num }}
num.recovery.threads.per.data.dir=1
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=1
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
zookeeper.connect={{ ZOOKEEPER_HOSTS }}/kafka
zookeeper.connection.timeout.ms=18000

```

```
group.initial.rebalance.delay.ms=0  
delete.topic.enable=true  
default.replication.factor=3
```

Listing 5.3: Przykładowy plik konfiguracyjny dla Kafki, korzystający z narzędzia do szablonowania Jinja2.

5.2 Orkiestracja

Kolejnym etapem realizacji projektu, jest realizacja orkiestracji, w skład której wchodzi realizacja wcześniejszego założonego modelu danych oraz napisanie orkiestratora, który będzie typowym wzorcem "Fasada" dla naszego systemu. Będzie on udostępniał uproszczony interfejs, umożliwiający manipulację elementami naszego abstrakcyjnego modelu danych.

5.2.1 Realizacja modelu danych

Model danych został zrealizowany przy pomocy kilku tabel stworzonych w bazie danych HBase. Te tabele to:

- DATASETS - zbiory danych
- FLINK_DATAFLOWS - strumienie na klastrze Flink
- SPARK_DATAFLOWS - strumienie na klastrze Spark
- DATA_TABLES - tabele danych

Schematy powyższych tabeli są przedstawione na listingu 5.4.

```
class HBaseTables:  
    FLINK_JOBS = Table(  
        "FLINK_JOBS",  
        Column("job_name", "VARCHAR", primary_key=True, not_null=True),  
        Column("jobmanager_address", "VARCHAR", primary_key=True, not_null=  
              True),  
        Column("jar_id", "VARCHAR"),  
    )  
  
    DATASETS = Table(  
        "DATASETS",  
        Column("id", "VARCHAR", primary_key=True, not_null=True),  
        Column("name", "VARCHAR"),  
        Column("timestamp", "TIMESTAMP"),  
        Column("tombstone", "TIMESTAMP"),  
    )  
  
    FLINK_DATAFLOWS = Table(
```

```

    "FLINK_DATAFLOWS",
    Column("id", "VARCHAR", primary_key=True, not_null=True),
    Column("flink_run_id", "VARCHAR"),
    Column("dataset_id", "VARCHAR"),
    Column("timestamp", "TIMESTAMP"),
)

SPARK_DATAFLOWS = Table(
    "SPARK_DATAFLOWS",
    Column("id", "VARCHAR", primary_key=True, not_null=True),
    Column("pid", "INTEGER"),
    Column("host_id", "VARCHAR"),
    Column("dataset_id", "VARCHAR"),
    Column("data_table_id", "VARCHAR"),
    Column("timestamp", "TIMESTAMP"),
    Column("tombstone", "TIMESTAMP"),
)

DATA_TABLES = Table(
    "DATA_TABLES",
    Column("id", "VARCHAR", primary_key=True, not_null=True),
    Column("dataset_id", "VARCHAR"),
    Column("name", "VARCHAR"),
    Column("timestamp", "TIMESTAMP"),
    Column("tombstone", "TIMESTAMP"),
)

```

Listing 5.4: Fragment kodu opisujący tabele użyte w bazie HBase.

Jak widać w tabelach tych, w niektórych przypadkach, została zastosowana kolumna *tombstone*, czyli „kamień nagrobny”. Jest to typowe rozwiązania w technologiach Big Data i bazach danych. Wierszy w tabelach się nie usuwa (a przynajmniej nie usuwa się ich od razu), a jedynie zapisuje się informację o usunięciu. Służy to temu, by operacje na naszym zbiorze danych mogły być uznane za *idempotentne*, czyli takie, które mogą być powtarzane wielokrotnie, a które nie wpływają na ostateczny wynik działania.

W tym wypadku chodzi o zapewnienie, że informacja o usunięciu danego zbioru danych nie zginie w momencie usunięcia wiersza. Baza danych HBase nie posiada więzów integralności, w związku z czym, należy o te więzy zadbać samemu. W momencie, w którym dopuścilibyśmy możliwość dowolnego usuwania wierszy z tabeli, mogłoby się okazać, że usuniemy zbiór o jakimś identyfikatorze i w tym momencie nastąpi awaria orkiestratora. W momencie ponownego startu informacja o tym, że dany zbiór danych został usunięty zostałaby utracona i dane na dysku pozostałyby już na zawsze, a przynajmniej do czasu aż ktoś ręcznie nie usunąłby tych danych. Ewentualnie możliwe byłoby stworzenie odpowiedniego zadania, które skanowałoby wszystkie zbiory danych i usuwałoby te „pozostałości”, które są już nieobecne w bazie, jednakże takie rozwiązania bardzo mocno obciążałoby samą bazę danych, zakładając, że zbiorów danych byłoby bardzo dużo (czego można się spodziewać, biorąc pod uwagę, że mamy tutaj do czynienia z Big

Data).

Strumienie danych wejściowe są realizowane przez klaster Kafka + Flink. Flink jest odpowiedzialny za przenoszenie danych z klastra Kafka do klastra Hadoop. Klaster Kafka natomiast powinien być skonfigurowany tak, by samemu usuwał dane już nieaktualne z punktu widzenia systemu. Umożliwia mu to jego konfiguracja i parametry określające czas życia wiadomości na klastrze, ale również ilość dostępnego miejsca, które może być wykorzystane zanim wiadomości zostaną usunięte.

Strumienie wyjściowe są realizowane przez Spark Streaming, który może strumieniować dane zarówno wprost z klastra HDFS jak i z bazy danych HBase przy pomocy Apache Phoenix. Spark Streaming został wybrany przede wszystkim ze względu na lokalność węzłów YARN z których korzysta Spark, względem samych danych. Zarówno węzły HBase jak i HDFS znajdują się na tych samych maszynach co węzły YARN, które z kolei są obsługiwane przez Spark'a.

5.2.2 REST API

Dostęp do zdefiniowanego modelu danych odbywa się poprzez REST API napisanego we frameworku Flask w Pythonie. Poszczególne punkty dostępowe zdefiniowane we Flasku prezentują się na listingu 5.5.

```
api = Blueprint(
    "api.v1.datasets", __name__, url_prefix="/api/v1/datasets"
)

(...)

@api.route("/", methods=("GET",))
def datasets() -> Response:
    (...)

@api.route("/", methods=("POST",))
def dataset_add() -> Response:
    (...)

@api.route("/<dataset_id>", methods=("GET",))
def dataset_get(dataset_id: str) -> Response:
    (...)

@api.route("/<dataset_id>", methods=("DELETE",))
def dataset_delete(dataset_id: str) -> Response:
    return jsonify({})

@api.route("/<dataset_id>/output", methods=("GET",))
def dataset_outputs(dataset_id: str) -> Response:
    (...)

@api.route("/<dataset_id>/output", methods=("POST",))
def dataset_update(dataset_id: str) -> Response:
    (...)
```

```

def dataset_output_add(dataset_id: str) -> Response:
    (...)

@api.route("/<dataset_id>/output/<output_id>", methods=("GET",))
def dataset_output_get(dataset_id: str, output_id: str) -> Response:
    (...)

@api.route("/<dataset_id>/output/<output_id>", methods=("DELETE",))
def dataset_output_delete(dataset_id: str, output_id: str) -> Response:
    return jsonify({})

@api.route("/<dataset_id>/tables", methods=("GET",))
def dataset_tables(dataset_id: str) -> Response:
    (...)

@api.route("/<dataset_id>/tables", methods=("POST",))
def dataset_table_add(dataset_id: str) -> Response:
    (...)

@api.route("/<dataset_id>/tables/<table_id_or_name>", methods=("GET",))
def dataset_table_get(dataset_id: str, table_id_or_name: str) -> Response:
    (...)

@api.route("/<dataset_id>/tables/<table_id_or_name>", methods=("DELETE",))
def dataset_table_delete(
    dataset_id: str, table_id_or_name: str
) -> Response:
    return jsonify({})

@api.route(
    "/<dataset_id>/tables/<table_id_or_name>/output", methods=("GET",)
)
def dataset_table_outputs(
    dataset_id: str, table_id_or_name: str
) -> Response:
    (...)

@api.route(
    "/<dataset_id>/tables/<table_id_or_name>/output", methods=("POST",)
)
def dataset_table_output_add(
    dataset_id: str, table_id_or_name: str
) -> Response:
    (...)

@api.route(

```

```

"/<dataset_id>/tables/<table_id_or_name>/output/<output_id>",
methods=("GET",),
)
def dataset_table_output_get(
    dataset_id: str, table_id_or_name: str, output_id: str
) -> Response:
    (...)

@api.route(
    "/<dataset_id>/tables/<table_id_or_name>/output/<output_id>",
methods=("DELETE",),
)
def dataset_table_output_delete(
    dataset_id: str, table_id_or_name: str, output_id: str
) -> Response:
    return jsonify({})

```

Listing 5.5: Fragment kodu opisujący wszystkie punkty końcowe REST API.

W celu zachowania przejrzystości pominięto kod metod w przypadku gdy jest zaimplementowany. Jak widać w powyższym fragmencie kodu do tworzenia wszelkich abstrakcyjnych obiektów, czyli zbiorów danych, tabel i strumieni wyjściowych używane są metody typu POST. By uzyskać informację o danym obiekcie używa się metody GET, natomiast by usunąć dany obiekt, używa się metody DELETE. Aktualizacja stworzonych już raz obiektów nie jest przewidziana. By zaktualizować dany obiekt należy stworzyć nowy obiekt, natomiast stary należy usunąć.

W powyższym listingu widać że metody usuwania obiektów są puste, zwracają zawsze . Jest to spowodowane tym, że usuwanie obiektów w tym systemie jest o wiele trudniejsze niż ich dodawanie i z racji braku czasu i ograniczonych środków nie zostało zaimplementowane. Jednakże można pokrótko opisać w jaki sposób powinno zostać zaimplementowane.

Usuwanie obiektów

By usunąć dany obiekt, praktycznie w każdym wypadku, należy się posłużyć polem *tombstone*. Dzięki temu informacja o tym, że dany obiekt został usunięty nie zginie i będzie mogła być propagowana na inne powiązane elementy. W celu wprowadzenia skutecznego usuwania należałoby zaimplementować odpowiedniego demona, który monitorowałby bazę danych co jakiś czas w celu odnajdywania najnowszych tombstone'ów (tombstone ma format TIMESTAMP, czyli aktualnego czasu). Jeżeli taki tombstone zostałby wykryty musiałby być propagowany na inne elementy powiązane z danym obiektem (np. usunięcie zbioru danych wiążałoby się z usunięciem wszystkich powiązanych strumieni oraz tabel). Ponieważ posługujemy się tombstone'm to informacja o usunięciu nie zginie nawet w momencie awarii całego systemu i będzie mogła być kontynuowana w przypadku powrotu systemu ze stanu awarii do stanu normalnego działania. Zarówno strumienie wejściowe jak i wyjściowe posiadają swoje identyfikatory. Dzięki temu można jednoznacznie stwierdzić, czy dany działający proces strumienia na klastrze Flink'a

aby na pewno nie został już usunięty. Demon sprawdziłby wszystkie ostatnio zaktualizowane tombstone'y i sprawdził, czy powiązane z nimi strumienie wejściowe nie powinny już zostać usunięte, jeśli tak, powinien zostać wysłany sygnał zatrzymania takiego strumienia.

Nieco trudniej przedstawia się sytuacja w przypadku strumieni Spark'a. Strumienie Spark'a są uruchamiane wprost na maszynie orkiestratora (orkiestratorów) poprzez podproces (subprocess.Popen). Stąd w tabeli odnośnie strumieni Spark'a pojawia się identyfikator PID procesu. Jeśli dany proces o danym identyfikatorze działałby na maszynie orkiestratora, natomiast z ostatnio odczytanych tombstone'ów wynikałoby, że powinien zostać usunięty, to taki proces powinien zostać zatrzymany...

Ale czy na pewno?

Mogłoby się bowiem zdarzyć, że demon zleciłby zatrzymanie procesu, które zostałoby wykonane przez system, jednakże zaraz potem orkiestrator, tworząc nowy strumień danych, stworzyłby taki o takim samym identyfikatorze PID, jak niedawno zatrzymany proces. Jeśli demon jeszcze raz w przeciągu krótkiego czasu, skanowałby listę procesów, to znalazłby ten nowy proces i znowu go usunął, gdyż mylnie uznałby, że jest to stary proces strumienia wymagający usunięcia. Można by uznać zatem, że po wpisaniu tombstone'a i usunięciu procesu przez demona należałoby usuwać wiersz z tombstone'm tak, by każdy kolejny proces demona nie mógł znowu usunąć innego procesu o tym samym identyfikatorze PID. jednakże to także nie rozwiązuje problemu, gdyż awaria demona mogłaby się zdarzyć zaraz po zabiciu procesu strumienia i przed poprawnym usunięciem wiersza z tabeli.

By móc jednoznacznie identyfikować dane procesy, konieczne jest ustawianie w nich szczeźgólnej zmiennej środowiskowej o unikalnym identyfikatorze określonym w tabeli jako „id” strumienia. W przypadku obecnego projektu jest to zmienna SPARK_FLOW_UUID. Dzięki PID oraz narzędziu „psutil” można odczytać aktualny stan procesu i wartości jego zmiennych środowiskowych i jednoznacznie zidentyfikować dany proces jako wymagający zatrzymania, bądź nie.

5.2.3 Symulacja powstawania zbioru danych

Powstanie zbioru danych rozpoczyna się od zapytania POST wysyłanego pod adres: „/api/v1/datasets”. Zapytanie POST powinno posiadać schemę danego datasetu, czyli listę pól wraz z typami danych, jaki dany zbiór danych ma obsługiwać. Przykładowa zawartość takiego zapytania przedstawia się na listingu 5.6.

```
{  
    "name": "car_internal_data",  
    "schema": [  
        {  
            "column_name": "timestamp",  
            "column_type": "biginteger",  
            "nullable": false,  
            "primary_key": true,  
        },  
    ],  
}
```

```

{
    "column_name": "uuid",
    "column_type": "string",
    "nullable": true,
},
{
    "column_name": "car_id",
    "column_type": "string",
    "nullable": true,
},
{
    "column_name": "turn",
    "column_type": "float",
    "nullable": true,
},
{
    "column_name": "velocity",
    "column_type": "float",
    "nullable": true,
}
],
}

```

Listing 5.6: Zapytanie HTTP POST dla stworzenia zbioru danych.

Schema jest listą pól, dla każdego pola można określić atrybuty *primary_key*, *column_name*, *column_type*, *nullable*. W przypadku zdefiniowania większej ilości pól jako *primary_key* tworzony jest klucz wspólny na wszystkich tych polach. Po kluczu istnieje możliwość szybkiego przeszukiwania bazy danych. W bazie kolumnowej HBase w każdej tabeli musi istnieć klucz główny zdefiniowany przynajmniej na jednej kolumnie.

W momencie tworzenia zbioru danych tworzony jest wiersz w tabeli DATASETS w bazie HBase, który unikalnie i jednoznacznie identyfikuje dany zbiór danych. Również automatycznie tworzony jest strumień wejściowy na klastrze Flink, który automatycznie tworzy odpowiedni temat na klastrze Kafki. Z danego tematu na klastrze Kafki, klaster Flink'a będzie przesyłał dane do systemu HDFS.

Również w momencie tworzenia zbioru danych zapisywany jest schemat danego zbioru w klastrze HDFS w odpowiednim folderze. Schemat wykorzystywany jest później w tworzeniu tabel dla danego zbioru danych.

W momencie, w którym dane zostaną przesłane na dany temat w klastrze Kafki, zostaną automatycznie odczytane przez konsumenta w klastrze Flink'a, który następnie prześle je do klastra HDFS do odpowiedniego folderu. Od tego momentu będzie istniała możliwość tworzenia tabel dla danego zbioru danych, które są tymczasowym obrazem tego zbioru możliwym do odpytania przez odpowiednie zapytanie SQL. Również wobec zbioru danych na klastrze HDFS istnieje możliwość odpytanie przy użyciu zapytania SQL.

By stworzyć tabelę dla danego zbioru danych należy posłużyć się kolejnym zapytaniem

HTTP POST wysłanym na adres: „/api/v1/datasets/<dataset_id>/tables” widocznym na listingu 5.7.

```
{  
    "name": "table_some_data",  
    "schema": [  
        {  
            "column_name": "timestamp",  
            "column_type": "biginteger",  
            "nullable": false,  
            "primary_key": true,  
        },  
        {  
            "column_name": "car_id",  
            "column_type": "string",  
            "nullable": true,  
        },  
    ],  
    "select_expression": [  
        "timestamp AS id",  
        "car_id"  
    ]  
}
```

Listing 5.7: Zapytanie HTTP POST dla stworzenia tabeli.

Jak widać jest ono bardzo podobne do zapytania tworzenia zbioru danych, jedyna różnica to możliwość dodania nowej sekcji *select_expression*, która w tym wypadku jest odpowiedzialna za odpowiednią selekcję kolumn z pierwotnego zbioru danych. W podanym wyżej przykładzie kolumna ”timestamp” jest przemianowana na ”id” w nowej tabeli i dodatkowo wyciągnięta zostaje jedna kolumna o nazwie ”car_id”.

W momencie tworzenia tabeli tworzony jest odpowiedni wiersz w tabeli DATA_TABLES, który jednoznacznie identyfikuje daną tabelę. Później ten sam identyfikator jest wykorzystywany w przypadku strumieni wyjściowych dla tabel.

By zaprezentować wyniki przedstawionych operacji, czyli tworzenia zbioru danych i tworzenia odpowiednich tabel, należy wykonać kolejne zapytanie HTTP POST, które stworzy strumienie wyjściowe dla danych obiektów. Dla zbioru danych takie zapytanie musi zostać wysłane na adres „/api/v1/datasets/<dataset_id>/output” w postaci widocznej na listingu 5.8.

```
{  
    "select_expression": ["timestamp"]  
}
```

Listing 5.8: Zapytanie HTTP POST dla stworzenia strumienia wyjściowego dla zbioru danych.

Jak widać w tym wypadku nie istnieje już konieczność przesyłania schematu dla zbioru danych, gdyż został on już przesłany wcześniej, w momencie tworzenia tego zbioru.

W przypadku gdy potrzebne jest stworzenie strumienia wyjściowego dla tabeli należy wysłać puste zapytanie POST na adres: „/api/v1/datasets/<dataset_id>/tables/<table_id>/output”.

Założenie stworzonego systemu jest takie, że tabele prezentują dane w postaci porządkanej i nie wymagają dodatkowych argumentów. Wszelka konieczna obróbka danych jest przekazywana w momencie tworzenia tabel.

W przypadku każdego powyższego zapytania w odpowiedzi przesyłany jest odpowiedni identyfikator, który jednoznacznie identyfikuje tworzone obiekty. W przypadku tworzenia strumieni żaden identyfikator nie jest przesyłany, gdyż z punktu widzenia użytkownika są one nieważne, jednakże są one tworzone, gdyż są konieczne do identyfikacji procesów na danej maszynie odpowiedzialnej za stworzenie danego strumienia dla zbioru danych, czy też dla tabel. Wszystkie dane dotyczące strumieni danych są zapisywane w odpowiednich tabelach w bazie danych HBase.

Rozdział 6

Prezentacja wyników

W tym rozdziale zostaną zaprezentowane wyniki działania stworzonego systemu wraz ze zrzutami ekranu przedstawiającymi działające oprogramowanie.

6.1 Inicjalizacja

W momencie uruchomienia wszystkich klastrów można ujrzeć aplikacje diagnostyczne pod odpowiednimi adresami. Poniżej zaprezentowane są screeny z tych aplikacji (rysunki 6.1, 6.2, 6.3, 6.4):

The screenshot shows a web browser window titled "Namenode information - Falkon". The address bar indicates the URL is <http://192.168.2.201:9870/dfshealth.html#tab-overview>. The page has a green header bar with tabs: Hadoop (selected), Overview, Datanodes, Datanode Volume Failures, Snapshot, Startup Progress, Utilities. The main content area is divided into two sections: "Overview" and "Summary".

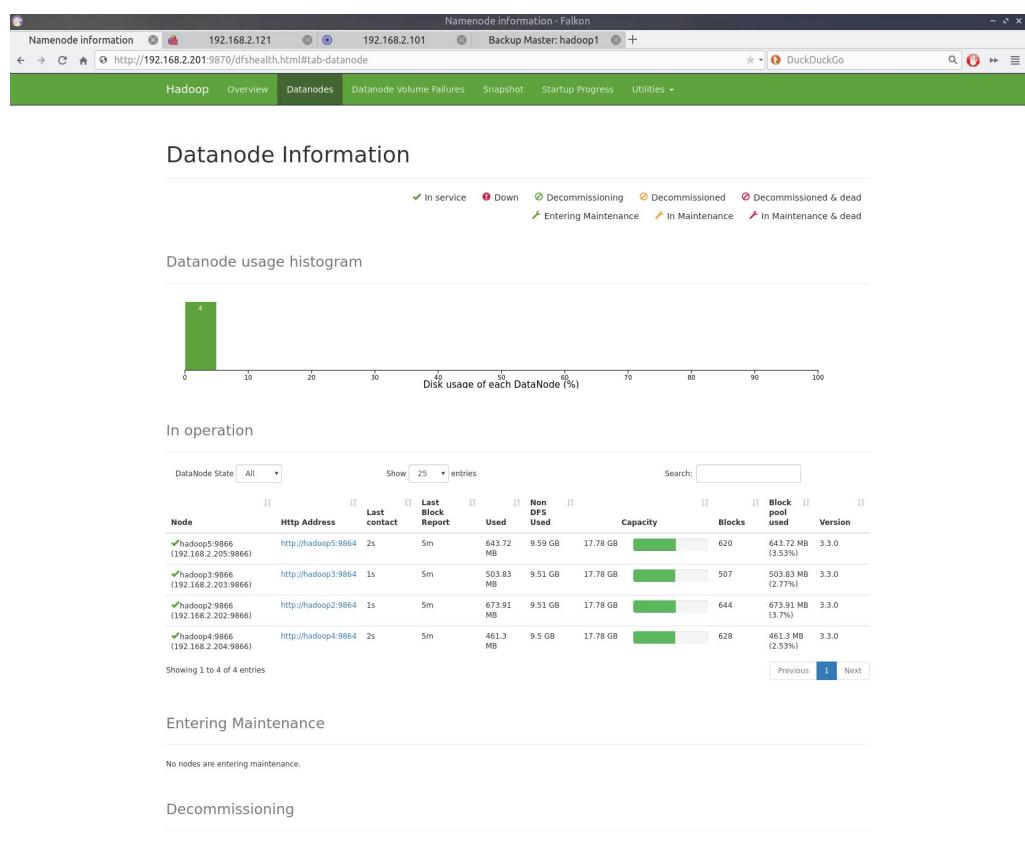
Overview section details:

Started:	Wed Aug 18 21:29:08 +0200 2021
Version:	3.3.0, raa96f1871bfb858f9baac59cf2a81ec470da649af
Compiled:	Mon Jul 06 20:44:00 +0200 2020 by brahma from branch-3:3.0
Cluster ID:	CID-84f5dc39-459f-43b6-89ad-6a223a0c09c54
Block Pool ID:	BP-62207334-192.168.2.201-1611352537681

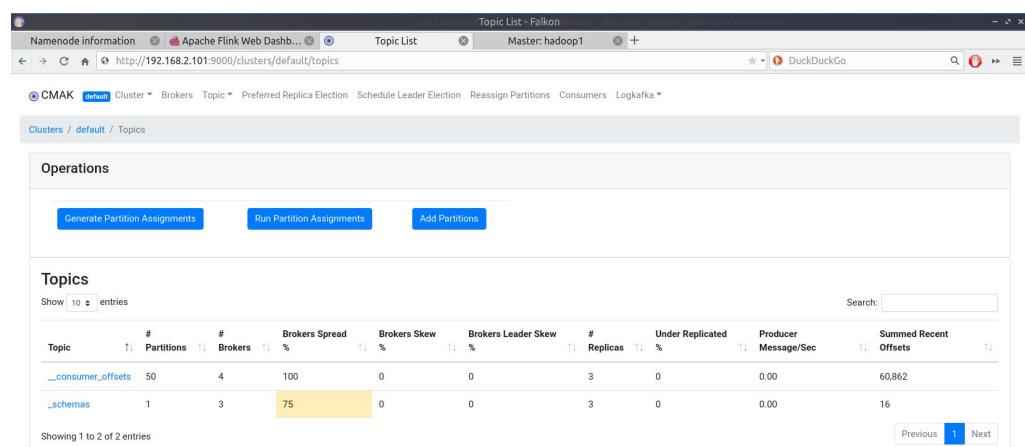
Summary section details:

Configured Capacity:	71.13 GB
Configured Remote Capacity:	0 B
DFS Used:	2.23 GB (3.13%)
Non DFS Used:	38.11 GB
DFS Remaining:	27.79 GB (39.07%)
Block Pool Used:	2.23 GB (3.13%)
DataNodes usages% (Min/Median/Max/stdDev):	2.53% / 3.53% / 3.70% / 0.49%
Live Nodes	4 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	0
Number of Blocks Pending Deletion (including replicas)	0
Block Deletion Start Time	Wed Aug 18 21:29:08 +0200 2021
Last Checkpoint Time	Wed Aug 18 21:29:12 +0200 2021
Enabled Erasure Coding Policies	RS-6-3-1024k

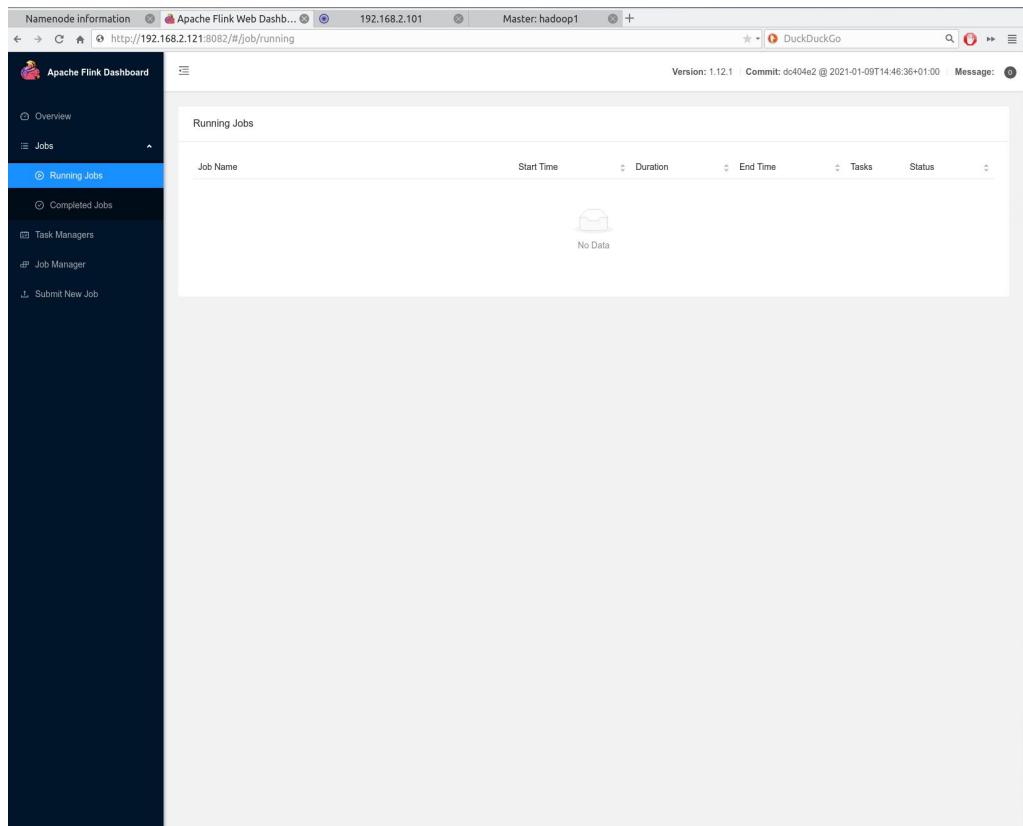
Rysunek 6.1: Aplikacja diagnostyczna Hadoop'a.



Rysunek 6.2: Diagnostyka węzłów Hadoop'a i zajętego miejsca.



Rysunek 6.3: Aplikacja diagnostyczna Kafki.



Rysunek 6.4: Aplikacja diagnostyczna Flink'a.

Spark, jeśli jest uruchamiany na klastrze YARN, nie posiada swojej aplikacji diagnostycznej, posiada ją tylko klaster YARN (Rys. 6.5).

The screenshot shows the Hadoop NodeManager UI at the URL <http://192.168.2.201:8088/cluster/nodes>. The left sidebar has sections for Cluster (About, Nodes, Node Labels, Applications, Scheduler - NEW, NEW_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, KILLED), Tools, and a search bar. The main content area is titled 'Nodes of the cluster'. It includes tabs for 'Cluster Metrics', 'Cluster Nodes Metrics', and 'Scheduler Metrics'. Below these are tables for 'Cluster Metrics' (with values like Apps Submitted: 0, Apps Pending: 0, etc.) and 'Scheduler Metrics' (with Scheduler Type: Capacity Scheduler). The bottom section shows a table of active nodes:

Node Labels	Rack	Node State	Node Address	Node HTTP Address	Last health-update	Health-report	Containers	Allocation Tags	Mem Used	Mem Avail	Vcores Used	Vcores Avail	Version
/default-rack	RUNNING	hadoop2:45454	hadoop2:8042		Thu Aug 19 22:49:31 +0200 2021		0		0 B	2 GB	0	2	3.3.0
/default-rack	RUNNING	hadoop4:45454	hadoop4:8042		Thu Aug 19 22:49:31 +0200 2021		0		0 B	2 GB	0	2	3.3.0
/default-rack	RUNNING	hadoop3:45454	hadoop3:8042		Thu Aug 19 22:49:33 +0200 2021		0		0 B	2 GB	0	2	3.3.0
/default-rack	RUNNING	hadoop5:45454	hadoop5:8042		Thu Aug 19 22:49:32 +0200 2021		0		0 B	2 GB	0	2	3.3.0

At the bottom, it says 'Showing 1 to 4 of 4 entries' and has navigation buttons: First, Previous, Next, Last. The status bar at the top right shows 'Logged in as: dr:who'.

Rysunek 6.5: Aplikacja diagnostyczna klastra YARN.

Na klastrze HBase automatycznie są tworzone podstawowe tabele konieczne do działania systemu zarządzania (rysunki 6.6 i 6.7).

The screenshot shows the Apache HBase master status interface. It includes:

- Region Servers**: A table listing 4 servers (hadoop2, hadoop3, hadoop4, hadoop5) with their start times, last contact, version, requests per second, and number of regions.
- Backup Masters**: A table showing 0 backup masters.
- Tables**: A table listing 15 tables across the default namespace. The columns include Namespace, Name, State, and Regions (OPEN, OPENING, CLOSED, CLOSING, OFFLINE, FAILED, SPLIT, Other). The table details section shows the following rows:

Namespace	Name	State	Regions	Description
default	DATASETS	ENABLED	1 OPEN	'DATASETS', {TABLE_ATTRIBUTES => [coprocessor\$1 => "org.apache.phoenix.coprocessor.ScanRegionObserver@805306366"], coprocessor\$2 => "org.apache.phoenix.coprocessor.UngroupedAggregateRegionObserver@805306366"}, coprocessor\$3 => "org.apache.phoenix.coprocessor.CachingEndpointImpl@805306366", coprocessor\$4 => "org.apache.phoenix.coprocessor.ServerCachingEndpointImpl@805306366", coprocessor\$5 => "org.apache.phoenix.index.IndexRegionObserver@805306366", coprocessor\$6 => "org.apache.hadoop.hbase.index.coders.ColumnValueIndexBuilder@805306366", coprocessor\$7 => "org.apache.hadoop.hbase.index.PhoenixIndexCodecs", coprocessor\$8 => "org.apache.phoenix.coprocessor.PhoenixTTLRegionObserver@805306364"}, {NAME => '0', DATA_BLOCK_ENCODING => 'FAST_DIFF', BLOOMFILTER => 'NONE', METADATA => 'NEW_VERSION_BEHAVIOR' => 'false'})
default	DATA_TABLES	ENABLED	1 OPEN	'DATA_TABLES', {TABLE_ATTRIBUTES => [coprocessor\$1 => "org.apache.phoenix.coprocessor.ScanRegionObserver@805306366"], coprocessor\$2 => "org.apache.phoenix.coprocessor.UngroupedAggregateRegionObserver@805306366"}, coprocessor\$3 => "org.apache.phoenix.coprocessor.GroupedAggregateRegionObserver@805306366", coprocessor\$4 =>

Rysunek 6.6: Aplikacja diagnostyczna HBase.

:: jdbc:phoenix:192.168.2.211> show tables;							
TABLE_CAT	TABLE_SCHEM	TABLE_NAME	TABLE_TYPE	REMARKS	TYPE_NAME	SELF_REFERENCING_COL_NAME	REF
	SYSTEM	CATALOG	SYSTEM TABLE				
	SYSTEM	CHILD_LINK	SYSTEM TABLE				
	SYSTEM	FUNCTION	SYSTEM TABLE				
	SYSTEM	LOG	SYSTEM TABLE				
	SYSTEM	MUTEX	SYSTEM TABLE				
	SYSTEM	SEQUENCE	SYSTEM TABLE				
	SYSTEM	STATS	SYSTEM TABLE				
	SYSTEM	TASK	SYSTEM TABLE				
		DATASETS	TABLE				
		DATA_TABLES	TABLE				
		FLINK_DATAFLOWS	TABLE				
		FLINK_JOBS	TABLE				
		SPARK_DATAFLOWS	TABLE				

Rysunek 6.7: Listing tabel z aplikacji konsolowej Apache Phoenix.

6.2 Stworzenie zbioru danych

W momencie w którym chcemy stworzyć zbiór danych, musimy wysłać odpowiednie pytanie do odpowiedniego adresu REST API. Do tego posłuży nam prosty skrypt w pythonie widoczny na listingu 6.1.

```
import requests

response = requests.post(
    "http://192.168.2.201:17777/api/v1/datasets", json={
        "name": "some_dataset",
        "schema": [
            {

```

```

        "column_name": "timestamp",
        "column_type": "biginteger",
        "nullable": False,
        "primary_key": True,
    },
    {
        "column_name": "my_column",
        "column_type": "string",
        "nullable": True,
    },
    {
        "column_name": "name",
        "column_type": "string",
        "nullable": True,
    }
],
}

print(response)
print(response.json())

```

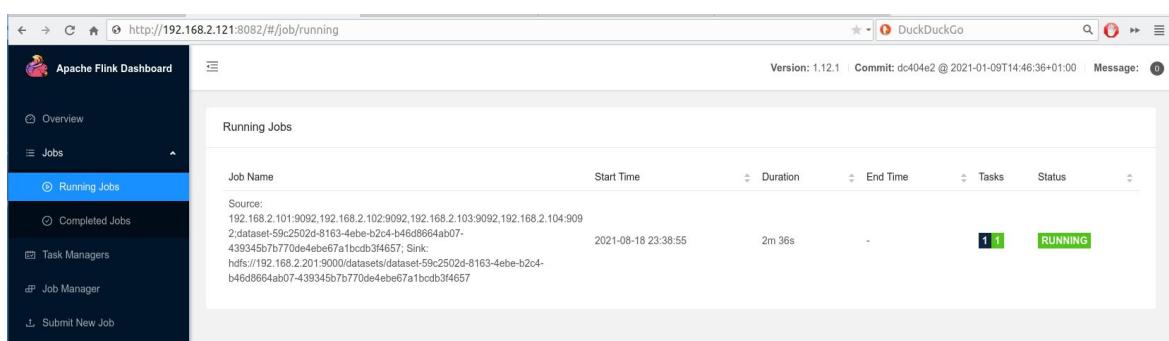
Listing 6.1: Skrypt napisany w Pythonie służący do stworzenia zbioru danych.

Jak widać serwer orkiestratora jest uruchomiony pod adresem 192.168.2.201:17777. Do obsługi całego REST API służy pythonowy serwer *waitress*. W momencie uruchomienia zapytania tworzony jest unikalny identyfikator zbioru danych, wysyłany w odpowiedzi widocznej na rysunku 6.8.

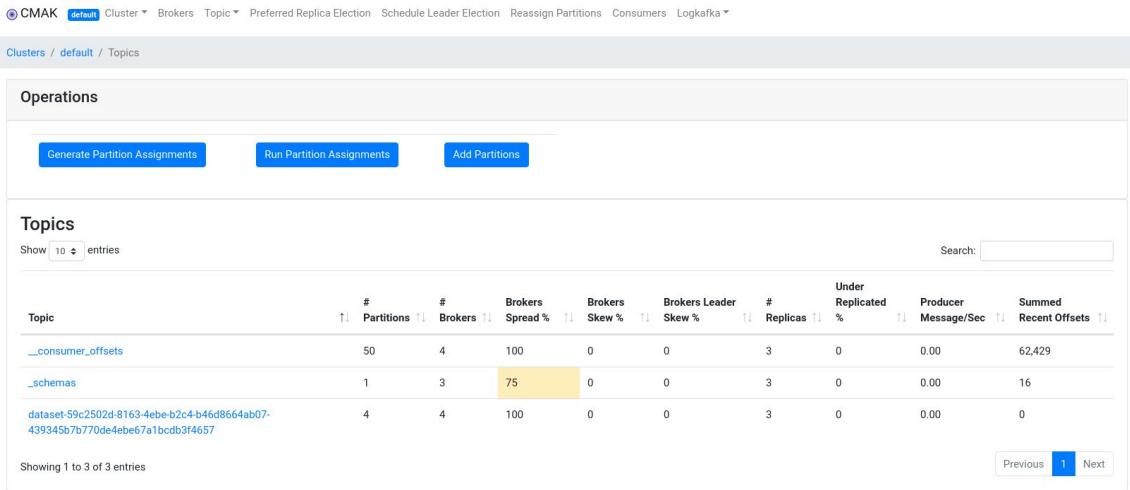
```
(python3.8) ptutak@ptutak-x570-stac:~/GitHub/av_decision_support_system/producers$ python post_dataset.py
<Response [200]>
{'response': [{"id': 'dataset-59c2502d-8163-4ebe-b2c4-b46d8664ab07-439345b7b770de4ebe67a1bcd3f4657', 'name': 'some_dataset'}], 'status': 'SUCCESS'}
(ptpython3.8) ptutak@ptutak-x570-stac:~/GitHub/av_decision_support_system/producers$
```

Rysunek 6.8: Odpowiedź serwera na zapytanie POST tworzące zbiór danych.

Następnie tworzony jest automatycznie strumień wejściowy na klastrze Flink'a, który automatycznie tworzy *topic* na klastrze Kafka o tym samym identyfikatorze co zbiór danych (rysunki 6.9 i 6.10).



Rysunek 6.9: Strumień wejściowy dla zbioru danych na klastrze Flink'a.



Rysunek 6.10: Stworzony temat na klastrze Kafki dla nowego zbioru danych.

Umożliwi to automatyczne przekazywanie wszelkich nowych danych wysyłanych do danego tematu od razu do klastra HDFS.

Do generowania danych został stworzony krótki skrypt w pythonie, który po uruchomieniu generuje dane w postaci CSV i wysyła je do klastra Kafki na stworzony temat nowego zbioru danych (Rys. 6.11).

Rysunek 6.11: Dane produkowane przez producenta do klastra Kafki.

Po uruchomieniu skryptu widoczny jest ruch na klastrze Kafki w temacie zbioru danych (Rys. 6.12). Również w liście konsumentów tematu danego zbioru danych widoczny jest konsument Flink'a dedykowany dla danego zbioru danych.

The screenshot shows the Kafka cluster management interface for the 'default' cluster. The top navigation bar includes links for Clusters, Brokers, Topic, Preferred Replica Election, Schedule Leader Election, Reassign Partitions, Consumers, and Logafka. The main content area displays the 'Topic Summary' and 'Operations' for the specified topic.

Topic Summary:

Replication	3
Number of Partitions	4
Sum of partition offsets	12
Total number of Brokers	4
Number of Brokers for Topic	4
Preferred Replicas %	100
Brokers Skewed %	0
Brokers Leader Skewed %	0
Brokers Spread %	100
Under-replicated %	0

Metrics:

Rate	Mean	1 min	5 min	15 min
Messages in /sec	0.70	1.08	1.17	1.19
Bytes in /sec	0.1k	0.1k	0.1k	0.1k
Bytes out /sec	5.29	18.82	4.18	1.42
Bytes rejected /sec	0.00	0.00	0.00	0.00
Failed fetch request /sec	0.00	0.00	0.00	0.00
Failed produce request /sec	0.00	0.00	0.00	0.00

Operations:

- Delete Topic
- Reassign Partitions
- Generate Partition Assignments
- Add Partitions
- Update Config
- Manual Partition Assignments

Partitions by Broker:

Broker	# of Partitions	# as Leader	Partitions	Skewed?	Leader Skewed?
1	3	1	(0,2,3)	false	false
2	3	1	(0,1,3)	false	false
3	3	1	(0,1,2)	false	false
4	3	1	(1,2,3)	false	false

Consumers consuming from this topic:

dataset-59c2502d-8163-4ebe-b2c4-b46d8664ab07-439345b7b770de4ebe67a1bcd3f4657	KF
--	----

Rysunek 6.12: Ruch danych na klastrze Kafki.

Do danego tematu zostało przesłanych 24 próbki danych, które są widoczne na ogólnym widoku tematów w klastrze Kafki (Rys. 6.13).

The screenshot shows the Kafka cluster management interface for the 'default' cluster. The top navigation bar includes links for Clusters, Brokers, Topic, Preferred Replica Election, Schedule Leader Election, Reassign Partitions, Consumers, and Logafka. The main content area displays the 'Topics' list.

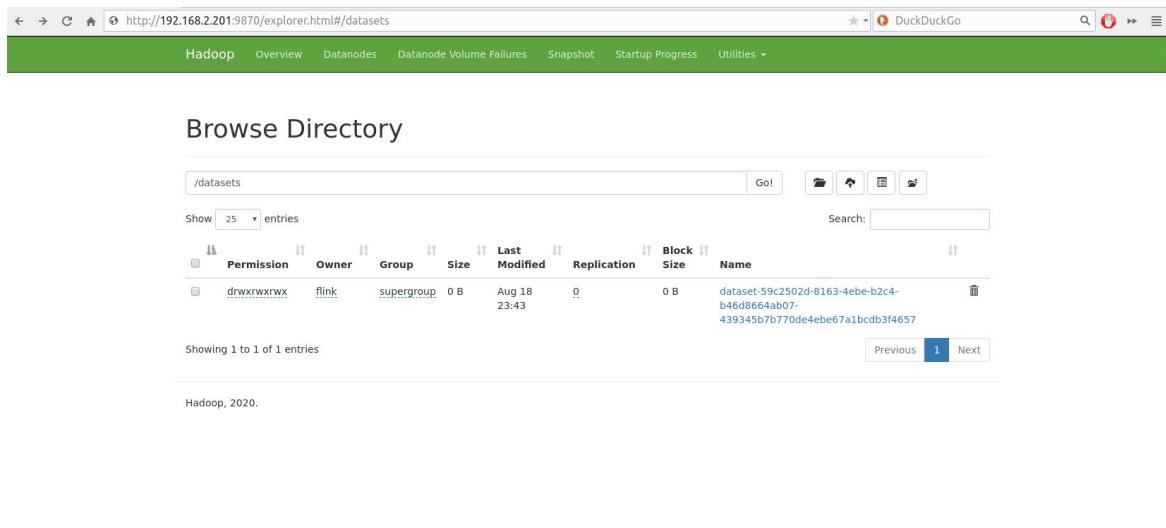
Topics:

Topic	# Partitions	# Brokers	Brokers Spread %	Brokers Skew %	Brokers Leader Skew %	# Replicas	Under Replicated %	Producer Message/Sec	Summed Recent Offsets
_consumer_offsets	50	4	100	0	0	3	0	0.80	62,478
_schemas	1	3	75	0	0	3	0	0.00	16
dataset-59c2502d-8163-4ebe-b2c4-b46d8664ab07-439345b7b770de4ebe67a1bcd3f4657	4	4	100	0	0	3	0	0.00	24

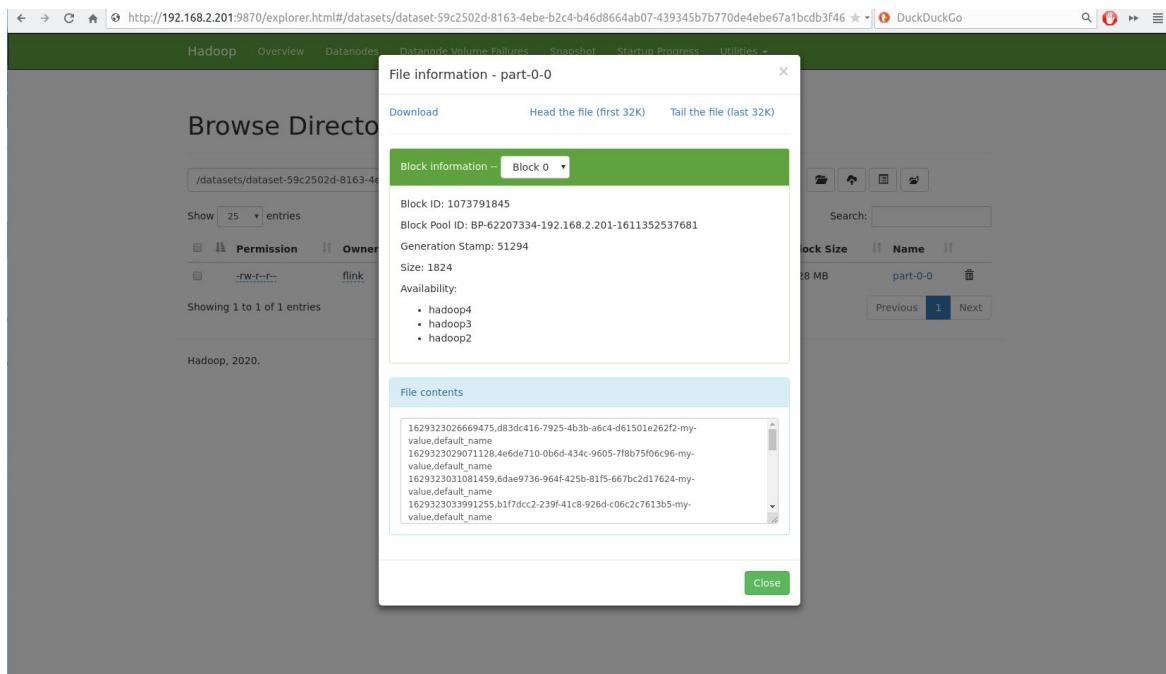
Showing 1 to 3 of 3 entries

Rysunek 6.13: Dane w temacie zbioru danych na klastrze Kafki.

Wszystkie te dane trafiły ostatecznie do klastra HDFS do odpowiedniego folderu z danymi (rysunki 6.14 i 6.15).



Rysunek 6.14: Folder z danymi na klastrze HDFS.



Rysunek 6.15: Podgląd danych na klastrze HDFS.

Powyższy przykład pokazuje jak można stworzyć zbiór danych oraz w jaki sposób dane są przesyłane do miejsca przechowywania danych.

6.3 Stworzenie tabel

By stworzyć tabele w systemie zarządzania danymi należy posłużyć się zapytaniem POST, które zostało zapisane w skrypcie widocznym na listingu 6.2.

```
import requests

response = requests.post(
```

```

    "http://192.168.2.201:17777/api/v1/datasets/dataset-59c2502d-8163-4ebe-
        b2c4-b46d8664ab07-439345b7b770de4ebe67a1bcd3f4657/tables",
    json={
        "name": "table_some_data",
        "schema": [
            {
                "column_name": "id",
                "column_type": "biginteger",
                "nullable": False,
                "primary_key": True,
            },
            {
                "column_name": "my_column",
                "column_type": "string",
                "nullable": True,
            }
        ],
        "select_expression": [
            "timestamp AS id",
            "my_column"
        ]
    }
)

print(response)
print(response.json())

```

Listing 6.2: Skrypt napisany w Pythonie służący do stworzenia tabeli.

Wynik działania skryptu uruchomionego w konsoli jest widoczny na rysunku 6.16.

```

[python3.8] ptutak@ptutak-x570-stac:[~/GitHub/av_decision_support_system/producer]$ python post_table.py
[response [200]
{'response': [{"data_table_id": 'Table_34ed1d0e_1a94_4858_8931_9b60ab1acd1e_04750496870e28bb37495fd9a8137766', 'dataset_id': 'dataset-59c2502d-8163-4ebe-b2c4-b46d8664ab07-439345b7b770de4ebe67a1bcd3f4657'}], 'status': 'SUCCESS'}
[python3.8] ptutak@ptutak-x570-stac:[~/GitHub/av_decision_support_system/producer]$ 

```

Rysunek 6.16: Odpowiedź orkiestratora w konsoli.

Jak widać nowo tworzonej tabeli został nadany odpowiedni identyfikator. Tabela o tym identyfikatorze została w tym momencie stworzona przez orkiestrator w bazie danych HBase (Rys. 6.17).

TABLE_CAT	TABLE_SCHEMA	TABLE_NAME	TABLE_TYPE	REMARKS	TYPE_NA
SYSTEM	CATALOG		SYSTEM TABLE		
SYSTEM	CHILD_LINK		SYSTEM TABLE		
SYSTEM	FUNCTION		SYSTEM TABLE		
SYSTEM	LOG		SYSTEM TABLE		
SYSTEM	MUTEX		SYSTEM TABLE		
SYSTEM	SEQUENCE		SYSTEM TABLE		
SYSTEM	STATS		SYSTEM TABLE		
SYSTEM	TASK		SYSTEM TABLE		
	DATASETS		TABLE		
	DATA_TABLES		TABLE		
	FLINK_DATAFLOWS		TABLE		
	FLINK_JOBS		TABLE		
	SPARK_DATAFLOWS		TABLE		
	TABLE_34ED1D0E_1A94_4858_8931_9B60AB1ACD1E_04750496870E28BB37495FD9A8137766		TABLE		

Rysunek 6.17: Widok nowo utworzonej tabeli w bazie danych HBase.

Również dla danej tabeli został uruchomiony strumień danych w Spark'u(Rys. 6.18).

ID	PID	HOST_ID	DATASET_ID	DATA_TABLE_ID	TIMESTAMP	TOMBSTONE
spark->707211-36c2-474c-9646-5d892c9f79-0e8d0f2800f7b61f0820c8023f9913d9 161098 192.168.2.201 dataset-59c250d0-8103-4ebe-b2c4-d46b664a607-439345d7b770de4eb97a10cc03f4637 table_34ed1d0e_1a94_4858_8931_9b60ab1acd1e_04750496870e28bb37495fd9a8137766 2021-08-19 01:46:34.802987 null						

Rysunek 6.18: Strumień danych w Spark'u dla nowo utworzonej tabeli.

Ten strumień danych to tak naprawdę zadanie przetwarzania wsadowego, które tworzy *snapshot* danych i zapisuje je do tabeli w bazie danych HBase. Nic nie stoi na przeszkodzie, by stworzyć działający strumień danych oparty o moduł Spark Streaming, bądź też okresowo uruchamiane zadanie wsadowe, które co jakiś czas aktualizuje dane w tabeli HBase podobnie jak dzieje się to w architekturze Lambda.

Po jakimś czasie odpowiednio zmodyfikowane dane są już widoczne w bazie HBase (Rys. 6.19).

ID	MY_COLUMN
1629323026669475	d83dc416-7925-4b3b-a6c4-d61501e262f2-my-value
1629323029071128	4e6de710-0b6d-434c-9605-7f8b75f06c96-my-value
1629323031081459	6dae9736-964f-425b-81f5-667bc2d17624-my-value
1629323033991255	b1f7dcc2-239f-41c8-926d-c06c2c7613b5-my-value
1629323036000761	10e15a82-c185-452c-9256-79aadbeb3796-my-value
1629323038010535	bf3d3038-ff1b-4941-83c8-f925d86ddf67-my-value
1629323040711792	80b9a247-39ce-4b75-9e60-e432cacd8406-my-value
1629323042722234	682f2012-3c44-4fdc-96b2-7239fb6ceb66-my-value
1629323044732148	c397847f-3f81-41d7-83c9-db339777c331-my-value
1629323046741808	a3141d3f-c44e-4ed5-95da-aaef9bc5eb70-my-value
1629323048754335	263473da-a4f2-4f0f-987f-2ece5b88cf9e-my-value
1629323050765312	f1972391-b876-4d4b-b200-2d1283ba57be-my-value
1629323052775513	f2a9df41-0827-4f11-a664-07d409044f0b-my-value
1629323054785527	e9e2e0d5-e970-4bf4-9311-998d367291ee-my-value
1629323056795679	ab816ee2-dcca-4497-a2b9-3c921d9e4784-my-value
1629323058805235	c72bd1f3-2dd8-488a-ab5e-64fdaba7fb5a-my-value
1629323060815228	5cfbcff8-0e3a-4fe0-b490-b2ecb2d4f13e-my-value
1629323062825362	a7bb6dd4f-82a8-4ea5-abbo-3d67f07a38e8-my-value
1629323064835620	1d4443e0-61c9-4326-a42e-1a6e038ecc42-my-value
162932306847193	a12e8765-4b12-4d2d-acb9-7ff0ab42e88-my-value
1629323069603555	f61c0ae0-8998-4129-8691-4e087a19a429-my-value
1629323071616402	6b83a1aa-28a6-424f-a317-1bebb2c4d999-my-value
1629323073626546	5a1e4fe4-bf09-4856-8c85-0ee9c9c48a54-my-value
1629323075638827	c4e96637-2722-438e-92a7-95543c13d814-my-value

Rysunek 6.19: Dane w bazie danych HBase.

Jak widać została poprawnie zmieniona nazwa pierwszej kolumny z *timestamp* na *id* oraz została pominięta ostatnia kolumna w danych, czyli *name*. Jak już zostało wspomniane istnieje bardzo dużo możliwości implementacji różnych operacji na danych, które mogłyby być później udostępniane poprzez REST API.

Jednakże dane zapisane w systemie są bezużyteczne, jeśli nie możemy ich w jakiś określony sposób odczytać. Jak już zostało wspomniane interfejsem wejścia/wyjścia w tworzonym systemie jest klaster Kafki, zatem jeśli dane mają być dostępne dla świata zewnętrznego, musi istnieć strumień wyjściowy, który przenosi te dane na klaster Kafki do nowo utworzonego tematu.

6.4 Stworzenie strumieni wyjściowych

Strumienie wyjściowe dla samego zbioru danych mogą być stworzone przy pomocy skryptu widocznego na listingu 6.3.

```

import requests

response = requests.post(
    "http://192.168.2.201:17777/api/v1/datasets/dataset-59c2502d-8163-4ebe-
        b2c4-b46d8664ab07-439345b7b770de4ebe67a1bcd3f4657/output",
    json={"select_expression": ["timestamp", "name"]}
)

print(response)
print(response.json())

```

Listing 6.3: Skrypt napisany w Pythonie służący do stworzenia strumienia wyjściowego dla zbioru danych.

Jak widać także w tym wypadku istnieje możliwość zastosowania dodatkowego filtru w postaci wyrażeń *select* lub też dowolnych innych wcześniej zaimplementowanych.

Odpowiedź, z orkiestratora w konsoli widoczna jest na rysunku 6.20.

```
(python3.8) ptutak@ptutak-x570-stac:~/github/av_decision_support_system/producers$ python post_output_dataset.py
<Response [200]>
{'response': [{"dataset_id": "dataset-59c2502d-8163-4ebe-b2c4-b46d8664ab07-439345b7b770de4ebe67a1bcd3f4657", "kafka_topic": "spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f"}], "status": "SUCCESS"}
(ptpython3.8) ptutak@ptutak-x570-stac:~/github/av_decision_support_system/producers$
```

Rysunek 6.20: Odpowiedź serwera dla zapytania POST tworzenia strumienia wyjściowego dla zbioru danych.

Jak widać w odpowiedzi podana jest nazwa tematu, która odpowiada identyfikatorowi stworzonego strumienia Spark'a na serwerze. Na klastrze Kafki tworzony jest odpowiedni temat (Rys. 6.21).

Topic	# Partitions	# Brokers	Brokers Spread %	Brokers Skew %	Brokers Leader Skew %	# Replicas	Under Replicated %	Producer Message/Sec	Summed Recent Offsets
_consumer_offsets	50	4	100	0	0	3	0	0.80	62,958
_schemas	1	3	75	0	0	3	0	0.00	16
dataset-59c2502d-8163-4ebe-b2c4-b46d8664ab07-439345b7b770de4ebe67a1bcd3f4657	4	4	100	0	0	3	0	0.00	24
spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f	4	4	100	0	0	3	0	0.00	24

Rysunek 6.21: Temat strumienia wyjściowego dla danych ze zbioru danych.

Jeżeli spróbujemy odczytać dane z tego tematu to otrzymamy poprawne wyniki, czyli dane wcześniej wygenerowane (Rys. 6.22).

```
(python3.8) ptutak@ptutak-x570-stac:~/GitHub/av_decision_support_system/producers$ python get_kafka_topic_data.py 192.168.0.10:9092
Assignment: [TopicPartition{topic=spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f,partition=0,offset=16664390568f3d5fb91ca11a6987dd0f}, TopicPartition{topic=spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 0 with key None: b'1629323026669475, default_name'}
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 1 with key None: b'1629323029071128, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 2 with key None: b'1629323031081459, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 3 with key None: b'1629323033991255, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 4 with key None: b'1629323036000761, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 5 with key None: b'1629323038010535, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 6 with key None: b'1629323040711792, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 7 with key None: b'1629323042722234, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 8 with key None: b'1629323044732148, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 9 with key None: b'1629323046741808, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 10 with key None: b'1629323048754335, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 11 with key None: b'1629323050765312, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 12 with key None: b'1629323052775513, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 13 with key None: b'1629323054785527, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 14 with key None: b'1629323056795679, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 15 with key None: b'16293230580805235, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 16 with key None: b'1629323060815228, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 17 with key None: b'1629323062823562, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 18 with key None: b'1629323064835620, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 19 with key None: b'1629323066847193, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 20 with key None: b'1629323069603555, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 21 with key None: b'1629323071616402, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 22 with key None: b'1629323073626546, default_name'
% spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e4390568f3d5fb91ca11a6987dd0f [0] at offset 23 with key None: b'1629323075638827, default_name'
```

Rysunek 6.22: Dane z tematu Kafki dla strumienia wyjściowego zbioru danych.

Również w tym wypadku nastąpiła poprawna filtracja i została usunięta jedna kolumna ze zbioru danych.

Tworzenie strumienia wyjściowego dla tabel jest bardzo podobne z tą różnicą, że nie można zdefiniować dla nich żadnych dodatkowych parametrów odnośnie filtrowania danych, zatem zapytanie POST wysyłane do API jest bardzo proste, zostało ono przedstawione na listingu 6.4.

```
import requests

response = requests.post(
    "http://192.168.2.201:17777/api/v1/datasets/dataset-59c2502d-8163-4ebe-b2c4-b46d8664ab07-439345b7b770de4ebe67a1bcd3f4657/tables/table_34ed1d0e_1a94_4858_8931_9b60ab1acd1e_04750496870e28bb37495fd9a8137766/output"
)

print(response)
print(response.json())
```

Listing 6.4: Skrypt napisany w Pythonie służący do stworzenia strumienia wyjściowego dla tabeli.

Odpowiedź na wywołane zapytanie jest bardzo podobna jak w przypadku strumienia wyjściowego dla zbioru danych (Rys. 6.23).

```
(python3.8) ptutak@ptutak-x570-stac:~/GitHub/av_decision_support_system/producers$ python post_table.py
Response: [200]
{
  "dataset": {
    "id": "table-59c2502d-8163-4ebe-b2c4-b46d8664ab07-439345b7b770de4ebe67a1bcd3f4657",
    "dataset_id": "dataset-59c2502d-8163-4ebe-b2c4-b46d8664ab07-439345b7b770de4ebe67a1bcd3f4657",
    "kafka_topic": "spark-32-bit-f5b-1-acb-4827-9ff9-4fa721d5903-10fb967a533fb59fe55410de97a1e1",
    "status": "SUCCESS"
}
}
```

Rysunek 6.23: Odpowiedź serwera dla zapytania POST tworzącego strumień wyjściowy tabeli.

Również w tym wypadku mamy podaną nazwę tematu na klastrze Kafki na którym pojawią się dane z tabeli, która jest jednocześnie nazwą stworzonego strumienia Spark'a w systemie (rysunki 6.24 i 6.25).

Topic	# Partitions	# Brokers	Brokers Spread %	Brokers Skew %	Brokers Leader	# Replicas	Under Replicated %	Producer Message/Sec	Summed Recent Offsets
__consumer_offsets	50	4	100	0	0	3	0	0.77	63,538
__schemas	1	3	75	0	0	3	0	0.00	16
dataset-59c2502d-8163-4ebe-b2c4-b46d8664ab07-439345b7b7770de4eb67a1bcd3f4657	4	4	100	0	0	3	0	0.00	24
spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6	4	4	100	0	0	3	0	0.00	24
spark-c4ec65ec-4b16-404d-b79e-136a64c4c816-d66e439056873d5fb91ca1fa6987dd0f	4	4	100	0	0	3	0	0.00	24

Rysunek 6.24: Lista tematów na klastrze Kafki wraz z tematem strumienia wyjściowego dla tabeli.

Również w tym wypadku dane odczytane z klastra Kafki odpowiadają tym zapisanym wcześniej w tabeli HBase:

```
(python3.8) ptutak@ptutak-x570-stac:~/GitHub/av_decision_support_system/producers$ python get_kafka_topic_data.py 192.168.2.100 topic=spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6,partition=0,offset=0,assignment=[TopicPartition{topic=spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6,partition=2,offset=1001,error=None},TopicPartition{topic=spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 0 with key None:b'1629323026669475,_083cd416-7925-4b3b-a6c4-061501e262f2-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 1 with key None:b'1629323029071128,4e6de710-0b6d-434c-9605-7fb875f06c96-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 2 with key None:b'1629323031081459,_6dae9736-964f-425b-81f5-667bc2d17624-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 3 with key None:b'1629323033991255,_b1f7dcc2-239f-41c8-926d-c06c2c7613b5-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 4 with key None:b'1629323036000761,10e15a82-c185-452c-9256-79aaedb3796-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 5 with key None:b'1629323038010535,_bfd33038-ff1b-4941-83ca-9f25d86d6f7-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 6 with key None:b'1629323040711792,_80b9a247-39ce-4b75-9e60-e432cadcd8406-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 7 with key None:b'1629323042722234,_682f2012-3c44-4fdc-96b2-7239fb6ceb6-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 8 with key None:b'1629323044732148,_c397847f-3f81-41d7-83c9-db339777c31-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 9 with key None:b'1629323046741808,_a3141d3f-c44e-4ed5-95da-aaef9bc5eb70-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 10 with key None:b'162932304754353,_263473da-a4f2-4f0f-987f-zece5b88cf9e-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 11 with key None:b'1629323050765312,_f1972391-b876-4d4b-b200-d21283ba70-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 12 with key None:b'1629323052775513,_f2a9df41-0827-4f11-aee6-07d4d9904f0b-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 13 with key None:b'1629323054785527,_e9e2e00d5-e970-4bf4-9311-998d367291ee-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 14 with key None:b'1629323056795679,_ab816ee2-dcca-4497-a2b9-3c921d9e4784-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 15 with key None:b'1629323058805235,_72bd1f3-2dd8-488a-ab5e-64fdaba7fb5a-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 16 with key None:b'1629323060815128,_5cfbcff8-0e3a-4ea0-b490-b2ecb2d4f13e-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 17 with key None:b'1629323062825362,_a7b66d4f-82a8-4ea5-abb0-3d67f07a38e3-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 18 with key None:b'1629323064835620,_1d4443e0-61c9-4326-a42e-1a6e038ecc42-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 19 with key None:b'1629323066847193,_a12e8765-4b12-4d2d-acb9-7ff0aba42e88-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 20 with key None:b'1629323069603555,_f61c0ae0-8998-4129-8691-4e087a19a29-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 21 with key None:b'1629323071616402,_6b83a1aa-28a6-424f-a317-1-bebb2c4d999-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 22 with key None:b'1629323073626546,_5a1e4fe4-bf09-4856-8c85-0ee9c9c48a54-my-value'%, spark-326b1f5b-5ac0-4827-9ffa-4fa621d55053-10d8067d63387a9590eb5410da97a1b6 [2] at offset 23 with key None:b'1629323075638827,_c4e96637-2722-438e-92a7-95543c13d814-my-value'
```

Rysunek 6.25: Dane odczytane z klastra Kafki ze strumienia wyjściowego dla tabeli.

Rozdział 7

Podsumowanie i wnioski

Niniejsza praca przedstawia efekty pracy mającej na celu stworzenie systemu zarządzania danymi dla pojazdów autonomicznych w oparciu o technologie Big Data.

Na początku pracy zostały przedstawione założenia teoretyczne tworzenia systemów rozproszonych oraz z jakimi problemami można się spotkać w przypadku próby stworzenia takich systemów. Następnie zostały przedstawione najważniejsze i najbardziej popularne architektury do przetwarzania dużych zbiorów danych oraz technologie używane do rozwiązywania problemów związanych z Big Data.

W kolejnej części pracy został przedstawiony projekt tworzonego systemu, a następnie została zaprezentowana realizacja projektu zarówno w części infrastrukturalnej jak i części tworzenia oprogramowania.

Założenia projektu zostały zrealizowane do końca, stworzony system działa poprawnie i realizuje wcześniej zdefiniowane zadania takie jak przechowywanie danych, katalogowanie i możliwość odczytywania danych.

Największą trudnością i częścią, która pochłonęła najwięcej czasu okazała się część infrastrukturalna, która zajęła ok 10 miesięcy. Oczywiście praca nad tą częścią nie odbywała się niestannie, jednakże w porównaniu z częścią dotyczącą oprogramowania była najbardziej uciążliwa i mozolna. By poprawnie ukończyć tę część musiało zostać stworzonych dziesiątki plików konfiguracyjnych, których opcje nie zawsze były dobrze opisane w dokumentacji. Etap ten często polegał na procesie prób i błędów, w którym to tworzona jest konfiguracja a następnie sprawdzane jest, czy działa ona poprawnie. Należy zaznaczyć że wykorzystywane narzędzia są same z siebie bardzo skomplikowane, posiadające niezliczone możliwości konfiguracji i stworzenie nawet podstawowej konfiguracji, która zapewnia poprawne działanie i jednocześnie umożliwia komunikację z innymi narzędziami nie jest proste.

Kolejną trudnością było stworzenie podstawowych zadań przetwarzania, odpowiedzialnych za podstawowe operacje na danych, takich jak przesyłanie danych z jednego narzędzia/klastra do drugiego. W teorii wszystkie używane narzędzia są ze sobą zintegrowane, w praktyce często ta integracja jest obecna tylko dla konkretnych wersji danych narzędzi, co czasem okazywało się dopiero po stworzeniu już całej infrastruktury dla jednego narzędzia i wymagało instalacji od nowa zupełnie nowej, starszej, bądź nowszej wersji tego samego narzędzia tylko po to, by

spełnić wymagania stawiane przez inne, używane już narzędzie.

Dużym wyzwaniem było też znajdowanie i naprawianie błędów. Używane narzędzia są pisane w Javie, która nie jest pierwszym językiem używanym przez autora tej pracy. Wszystkie komunikaty odnośnie błędów są najczęściej listingami z wyjątków rzucanych właśnie w języku Java.

Ilość pracy potrzebna do stworzenia poprawnie działającej infrastruktury zdecydowanie została niedoszacowana na początku tworzenia tego systemu, nawet przy bardzo dużym buforze zapasowym, jeśli chodzi o czas tworzenia pracy.

Tworzenie oprogramowania orkiestratora było zdecydowanie najprzyjemniejszą częścią pracy, gdyż wiele elementów, które w części strukturalnej były odrębnymi, niemającymi ze sobą związku, elementami, w tej części stawały się całością. Również cała podstawa infrastrukturalna została już zaimplementowana i umożliwiała poprawne działanie całego kodu orkiestratora.

Stworzony system ma duże możliwości rozwoju, szczególnie jeśli chodzi o udostępnione funkcje poprzez REST API oraz nie w pełni wykorzystane funkcje przetwarzania danych dostępne na klastrze Kafka i Flink'a. Należy zauważyć, że ta praca skupiała się przede wszystkim na możliwościach framework'u Spark + HDFS, jednakże istnieją ogromne możliwości rozwoju i sposobów przetwarzania danych w części Flink + Kafka, które z racji ograniczonego czasu nie były wykorzystane.

Bardzo duże możliwości rozwoju i optymalizacji tworzonego systemu drzemią również w konfiguracji wszystkich narzędzi. Stworzona do tej pory konfiguracja umożliwia poprawne działanie wszystkich elementów, jednakże nie nadaje się do środowisk produkcyjnych. Obszerna i odpowiednio dostosowana konfiguracja dla środowisk produkcyjnych z pewnością zwiększyłaby możliwości tworzonego systemu i zoptymalizowała działanie przede wszystkim pod kątem szybkości odpowiedzi i przetwarzania danych.

Kolejnym tematem, który nie był w żadnym stopniu zrealizowany w niniejszej pracy jest kwestia bezpieczeństwa, autentykacji i autoryzacji użytkowników. W tworzonym systemie z racji na ograniczone ramy czasowe nie było możliwości zaimplementowania choćby szczegółowej konfiguracji używanych narzędzi w dziedzinie bezpieczeństwa, choć wszystkie posiadają funkcje związane z zarządzaniem dostępami dla poszczególnych użytkowników.

Tematem ciekawym z teoretycznego punktu widzenia okazał się problem poprawnego zdefiniowania operacji dodawania i usuwania zbiorów danych, tabel i strumieni w środowisku rozproszonym.

Bibliografia

- [1] *Apache Storm Tutorial - Introduction*. URL: <https://www.simplilearn.com/introduction-to-storm-tutorial-video> (term. wiz. 19. 12. 2020).
- [2] Aniruddha Bhandari. *A Beginner's Guide to CAP Theorem for Data Engineering*. URL: <https://www.analyticsvidhya.com/blog/2020/08/a-beginners-guide-to-cap-theorem-for-data-engineering/> (term. wiz. 16. 12. 2020).
- [3] Matei Zaharia Bill Chambers. *Spark: The Definitive Guide: Big Data Processing Made Simple*. Sebastopol: O'Reilly Media, 2018.
- [4] Nikola Bunović. *30 big data statistics everybody's taling about*. URL: <https://dataprof.net/statistics/data-statistics/#:~:text=It%20estimated%20that%20around%202.5,every%20individual%20on%20the%20planet.> (term. wiz. 12. 12. 2020).
- [5] Jeff Desjardins. *How much data is generated each day?* URL: <https://www.weforum.org/agenda/2019/04/how-much-data-is-generated-each-day-cf4bddf29f/> (term. wiz. 12. 12. 2020).
- [6] *Flink Tutorial – A Comprehensive Guide for Apache Flink*. URL: <https://data-flair.training/blogs/flink-tutorial/> (term. wiz. 19. 12. 2020).
- [7] Keith D. Foote. *A Brief History of the Hadoop Ecosystem*. URL: <https://www.dataversity.net/a-brief-history-of-the-hadoop-ecosystem/#> (term. wiz. 12. 12. 2020).
- [8] Amir Kalron. *How do Hadoop and Spark Stack Up?* URL: <https://logz.io/blog/hadoop-vs-spark/> (term. wiz. 18. 12. 2020).
- [9] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable and Maintainable Systems*. Sebastopol: O'Reilly Media, 2017.
- [10] Jay Kreps. *Questioning the Lambda Architecture*. URL: <https://www.oreilly.com/radar/questioning-the-lambda-architecture/> (term. wiz. 17. 12. 2020).
- [11] Nathan Marz. *How to beat the CAP theorem*. URL: <http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html> (term. wiz. 17. 12. 2020).
- [12] James Warren Nathan Marz. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Shelter Island: Manning Publications, 2015.

- [13] Todd Palino Neha Narkhede Gwen Shapira. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. Sebastopol: O'Reilly Media, 2017.
- [14] Jack Vanlightly. *RabbitMQ vs Kafka Part 1 - Two Different Takes on Messaging*. URL: <https://jack-vanlightly.com/blog/2017/12/4/rabbitmq-vs-kafka-part-1-messaging-topologies> (term. wiz. 19. 12. 2020).
- [15] Tom White. *Hadoop: The Definitive Guide: Storage and Analysis at Internet Scale*. Sebastopol: O'Reilly Media, 2015.