NIERELACYJNE ROZWIĄZANIA BAZODANOWE

WYKŁAD 7

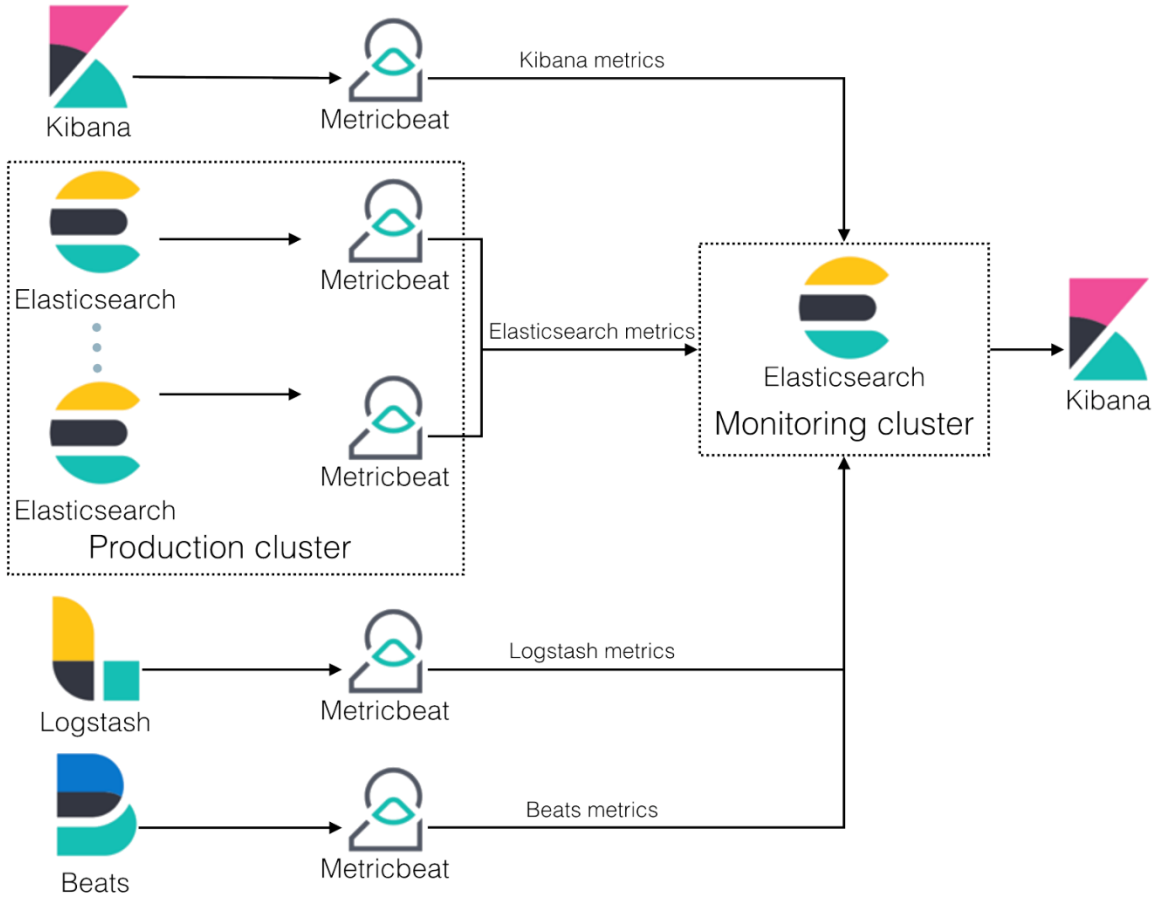
AGENDA

- Baza ElasticSearch
- Różnice w bazach NoSQL
- Apache Spark

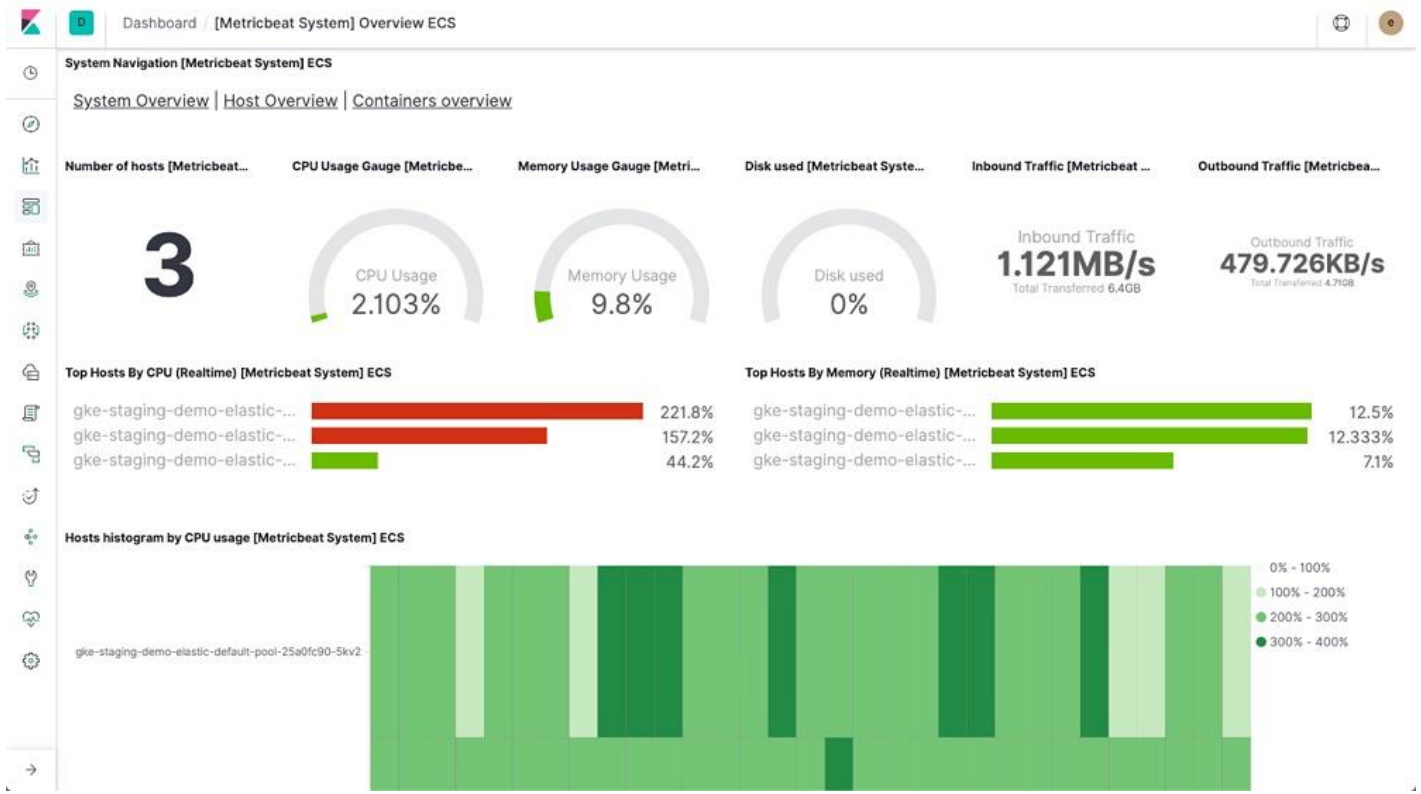
ELASTICSEARCH

- Baza NoSQL
- Zbliżona do MongoDB - działa na dokumentach JSON
- Szybsza i wydajniejsza niż MongoDB
- Zapytania definiowane w JSON

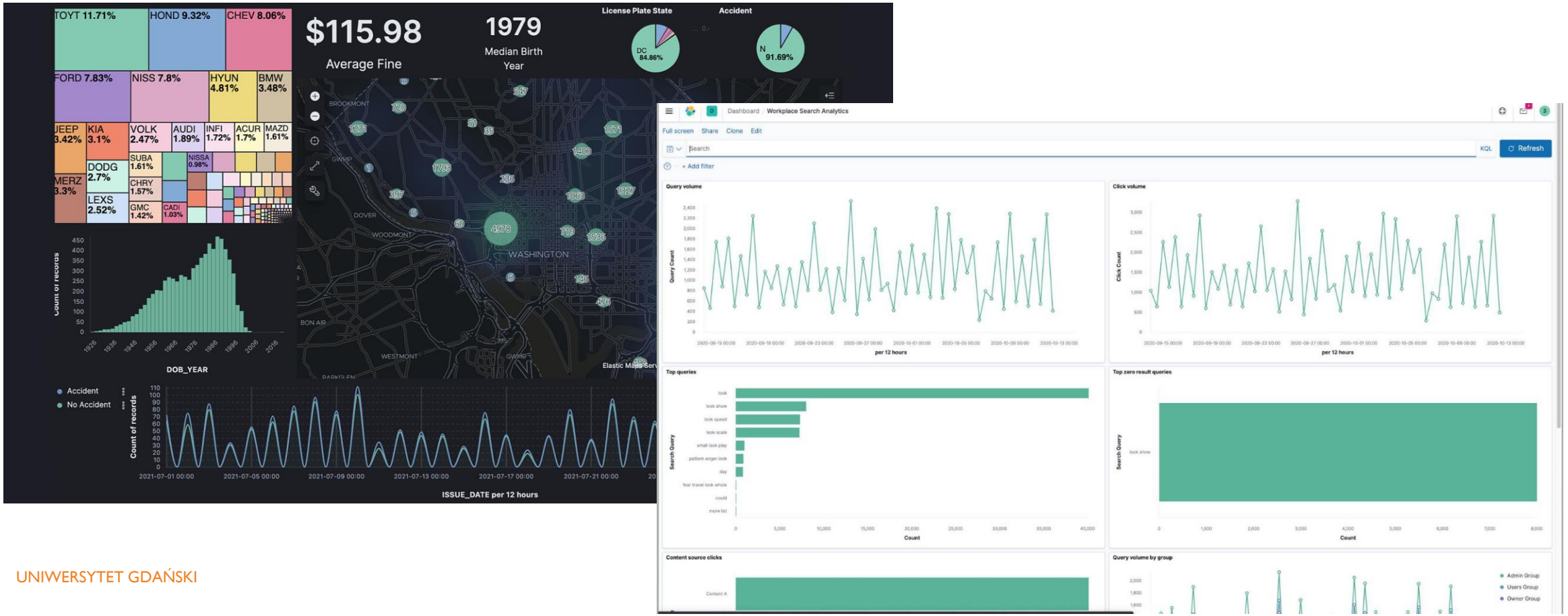
ARCHITEKTURA (DOKUMENTACJA ELASTICSEARCH)



METRICBEAT (DOKUMENTACJA ELASTICSEARCH)



KIBANA (DOKUMENTACJA ELASTICSEARCH)



CHARAKTERYSTYKA ELASTICSEARCH

- 1. Ogólna charakterystyka
  - Typ: Otwarta, rozproszona wyszukiwarka.
  - Podstawowa funkcjonalność: Wyszukiwanie pełnotekstowe, wyszukiwanie danych strukturalnych i niestrukturalnych oraz analiza.
  - Podstawowy silnik: Zbudowany na Apache Lucene.
  - Interfejs: RESTful API do interakcji i zapytań.

CHARAKTERYSTYKA ELASTICSEARCH

- 2. Architektura
  - Oparta na klastrze: Elasticsearch działa jako klaster węzłów.
  - Typy węzłów:
  - Węzeł główny: Zarządza stanem klastra i metadanymi.
  - Węzeł danych: Przechowuje dane i przetwarza zapytania.
  - Węzeł pobierania: Wstępnie przetwarza dokumenty przed indeksowaniem.
  - Węzeł koordynatora: Równoważy dystrybucję zapytań w klastrze.
  - Skalowanie poziome: Dane są dystrybuowane w węzłach za pomocą fragmentów.
  - Replikacja: Obsługuje wiele replik w celu zapewnienia tolerancji błędów i wysokiej dostępności.
  - Skalowanie elastyczne: Węzły można dodawać lub usuwać dynamicznie bez przestojów.

CHARAKTERYSTYKA ELASTICSEARCH

- 3. Indeksowanie i przechowywanie
  - Indeksowanie: Przechowuje dokumenty w formacie JSON. Inverted Index: podstawowa struktura danych zoptymalizowana pod kątem wyszukiwania pełnotekstowego.
  - Shards i Replicas:
    - Primary Shards: podpodziały indeksu.
    - Replica Shards: kopie primary shards w celu zapewnienia redundancji.
  - Schema:
    - elastyczny schemat z opcjonalnymi mapowaniami dla typów pól.
    - Obsługuje dynamiczne wykrywanie pól i wnioskowanie o typach.

CHARAKTERYSTYKA ELASTICSEARCH

- 4. Możliwości zapytań
  - Query DSL (Domain-Specific Language):
  - Obsługuje zapytania boolowskie (must, should, must_not).
  - Zapytania pełnotekstowe, terminowe, zakresowe, rozmyte i wieloznaczne.
  - Aggregation Framework:
  - Zapewnia metryki (np. średnia, suma) i agregacje kubełkowe (np. terminy, histogram).
  - Funkcje wyszukiwania:
  - Ocena trafności: wykorzystuje algorytmy TF-IDF i BM25 do klasyfikowania wyników.
  - Autouzupełnianie i sugestie: przewidywanie tekstu i tolerancji literówek.
  - Wyszukiwanie w pobliżu: Znajduje terminy w określonej odległości w tekście.

CHARAKTERYSTYKA ELASTICSEARCH

- 5. Optymalizacja wydajności
  - Buforowanie: Obejmuje bufory danych zapytań i pól.
  - Optymalizacja fragmentów: Równoważy dane w fragmentach w celu równoległego wykonywania zapytań.
  - Interwał odświeżania: Kontroluje, jak szybko dokumenty są przeszukiwalne po indeksowaniu.
  - Kompresja: Używa LZ4 i DEFLATE w celu wydajnego przechowywania danych.

CHARAKTERYSTYKA ELASTICSEARCH

- 6. Skalowalność
  - Skalowalność pozioma: Dystrybuuje indeksy w wielu węzłach i regionach.
  - Koordynacja klastra:
  - Zen Discovery: Zapewnia koordynację węzłów i tolerancję błędów.
  - Automatyczne przełączanie awaryjne na sprawne węzły.

CHARAKTERYSTYKA ELASTICSEARCH

- 7. Integracja i ekosystem
  - Elastic Stack:
  - Kibana: Narzędzie do wizualizacji i pulpitu nawigacyjnego dla Elasticsearch.
  - Logstash: Potok pobierania danych dla danych dziennika i zdarzeń.
  - Beats: Lekkie agenty do wysyłania danych do Elasticsearch.
  - Integracja z innymi systemami:
  - Natywne łączniki dla Hadoop, Kafka i różnych baz danych. REST API i SDK dla Pythona, Java, Go itp.

CHARAKTERYSTYKA ELASTICSEARCH

- 8. Możliwości w czasie rzeczywistym
  - Near Real-Time (NRT): Aktualizacje indeksowania i wyszukiwania są widoczne niemal natychmiast.
  - Strukturyzowane przesyłanie strumieniowe: Przetwarza dzienniki i zdarzenia w czasie rzeczywistym za pomocą Logstash i Beats.
- 9. Bezpieczeństwo
  - Elastyczne zabezpieczenia (funkcje komercyjne i typu open source):
  - Kontrola dostępu oparta na rolach (RBAC).
  - Szyfrowana komunikacja za pomocą TLS/SSL.
  - Uwierzytelnianie i autoryzacja za pomocą natywnych domen lub systemów zewnętrznych (LDAP, OAuth).
  - Rejestrowanie audytu: Śledzi wszystkie działania użytkowników pod kątem zgodności.

CHARAKTERYSTYKA ELASTICSEARCH

- 10. Monitorowanie i zarządzanie
  - Narzędzia monitorujące:
  - Wbudowane monitorowanie za pomocą Kibany.
  - Integracja z zewnętrznymi systemami monitorowania, takimi jak Prometheus i Grafana.
  - Interfejsy API:
  - Stan i kondycja klastra.
  - Statystyki na poziomie węzła i fragmentu.
- 11. Opcje wdrażania
  - Lokalnie: Wdrażanie w lokalnej infrastrukturze.
  - Chmura: Elastic Cloud, AWS, Azure, obsługa GCP.
  - Konteneryzacja: obsługuje wdrożenia Docker i Kubernetes.

CHARAKTERYSTYKA ELASTICSEARCH

- 12. Licencjonowanie
  - Elastyczna licencja 2.0: obejmuje funkcje open source i premium.
  - OpenSearch: open source przez AWS.
- 13. Obsługiwane typy danych
  - Podstawowe typy danych:
  - Tekst, słowo kluczowe, liczba (liczba całkowita, liczba zmiennoprzecinkowa), wartość logiczna, data.
  - Zaawansowane typy danych:
  - Geo-punkt, geo-kształt, IP i obiekty zagnieżdżone.

CHARAKTERYSTYKA ELASTICSEARCH

- 14. Zastosowanie
- Wyszukiwanie pełnotekstowe: wyszukiwarki, katalogi e-commerce.
- Monitorowanie dzienników i zdarzeń: scentralizowane rejestrowanie i śledzenie błędów.
- Analiza danych: analiza w czasie rzeczywistym i wsadowa.
- Analiza bezpieczeństwa: wykrywanie zagrożeń i dochodzenia kryminalistyczne.
- 15. Ograniczenia
- Spójność: zoptymalizowane pod kątem ostatecznej spójności.
- Intensywne wykorzystanie pamięci: duże wykorzystanie pamięci w przypadku operacji na dużą skalę. Obciążenie indeksowania: Tworzenie indeksu może spowolnić aktualizacje w czasie rzeczywistym.

ELASTICSEARCH – PRZYKŁAD ZAPYTANIA SELECT W JĘZYKU PYTHON

```
from elasticsearch import Elasticsearch

es = Elasticsearch("http://localhost:9200")

index_name = "moj_indeks"
query = {
    "query": {
        "match": {
            "title": "szukane wyrazenie"
        }
    }
}
try:
    response = es.search(index=index_name, body=query)

    print("Wyniki:")
    for hit in response["hits"]["hits"]:
        print(f"ID: {hit['_id']}, Score: {hit['_score']}, Source: {hit['_source']}")

except Exception as e:
    print(f"Blad: {e}")
```

OGÓLNE PRZEZNACZENIE

	MongoDB	Elasticsearch
Główne zastosowanie	Uniwersalna baza danych NoSQL do danych aplikacyjnych.	Silnik wyszukiwania zoptymalizowany pod kątem wyszukiwania pełnotekstowego i analityki.
Model danych	Zorientowany na dokumenty (dokumenty BSON podobne do JSON).	Zorientowany na indeksy (dokumenty JSON przechowywane w odwróconych indeksach).

ARCHITEKTURA

	MongoDB	Elasticsearch
Indeksowanie	Opcjonalne indeksy wtórne.	Automatyczne indeksowanie wszystkich pól dla szybkiego wyszukiwania.
Format przechowywania	BSON (Binary JSON).	JSON (przechowywany w odwróconych indeksach).
Skalowanie	Skalowanie horyzontalne z użyciem shardingu.	Skalowanie horyzontalne z użyciem shardów i replik.
Replikacja	Zestawy replik dla wysokiej dostępności.	Węzły replikujące dla tolerancji błędów i równoważenia obciążenia.
Schemat	Elastyczny schemat (bez schematu, opcjonalna walidacja).	Brak schematu, ale definicje mapowań optymalizują wydajność.

ZAPYTANIA

	MongoDB	Elasticsearch
Język zapytań	MongoDB Query Language (MQL).	Elasticsearch Query DSL (Domain-Specific Language).
Wyszukiwanie	Podstawowe i złożone zapytania z ograniczonym wyszukiwaniem tekstowym.	Zaawansowane wyszukiwanie pełnotekstowe, ocena trafności i ranking.
Agregacja	Framework agregacji do przetwarzania i transformacji danych.	Obsługuje agregacje z operacjami metrycznymi i kubełkowymi.

WYDAJNOŚĆ

	MongoDB	Elasticsearch
Pobieranie danych	Zoptymalizowane pod kątem operacji CRUD na dokumentach.	Zoptymalizowane pod kątem wyszukiwania i analityki z niskimi opóźnieniami.
Szybkość wyszukiwania	Relatywnie wolniejsze dla złożonych zapytań tekstowych.	Bardzo szybkie dla wyszukiwania pełnotekstowego i wyników rankingowych.

PRZECHOWYWANIE DANYCH I ZARZĄDZANIE

	MongoDB	Elasticsearch
Trwałość danych	Trwałe przechowywanie jako główna baza danych.	Trwałe lub tymczasowe przechowywanie danych dla indeksowania.
Wolumen danych	Obsługuje duże zestawy danych, zoptymalizowany dla danych operacyjnych.	Zoptymalizowany dla wyszukiwania i analityki z dużymi indeksami.

ZASTOSOWANIE

MongoDB	Elasticsearch
Uniwersalna baza danych aplikacyjnych.	Analiza i monitorowanie logów.
Przechowywanie dokumentów podobnych do JSON i zarządzanie danymi transakcyjnymi.	Wyszukiwanie pełnotekstowe i zapytania rankingowe.
Przechowywanie i pobieranie danych operacyjnych w czasie rzeczywistym.	Silniki wyszukiwania w e-commerce, systemy rekomendacji.
Ewolucja schematu i elastyczne modele danych.	Indeksowanie i analiza danych strukturalnych/nieustrukturalnych.

SKALOWALNOŚĆ

	MongoDB	Elasticsearch
Skalowanie horyzontalne	Shardy dla rozproszonych baz danych.	Shardy dla rozproszonego wyszukiwania i indeksowania.
Wydajność zapisu	Wysoka przepustowość zapisu dla operacji na dokumentach.	Operacje zapisu są wolniejsze z powodu narzutu indeksowania.

KOSZTY

	MongoDB	Elasticsearch
Licencja	SSPL (Server Side Public License) dla nowszych wersji.	Elastic License 2.0 dla nowszych wersji.
Usługi w chmurze	MongoDB Atlas (usługa hostowana).	Elastic Cloud (usługa hostowana).

APACHE SPARK

SZYBKIE POWTÓRZENIE

- Hadoop
- Spark
- Kafka
- Storm
- Scala
- Hive
- MapReduce
- HBase
- Cassandra
- Python

WERSJE APACHE SPARK

- Apache Spark to rozproszony silnik przetwarzania danych.

- Lata 2009 (UC Berkley) -> 2013 (Apache)
- 2012 – wersja 0.5
- 2014 – wersja 1.0
- 2016 – wersja 2.0
- 2020 – werjsa 3.0
- 2024 – wersja 3.5.3

- Scala
- Python
- Java
- R

- Katalog /spark/bin

PODSTAWOWE CECHY APACHE SPARK

- 1. Ogólna architektura
  - Cluster Computing Framework: Używa architektury master-slave z programem sterownika koordynującym zadania w wielu węzłach roboczych.
  - Unified Engine: Obsługuje różne obciążenia, w tym przetwarzanie wsadowe, zapytania interaktywne, przesyłanie strumieniowe danych i uczenie maszynowe.
- 2. Podstawowe komponenty
  - Spark Core: Zapewnia podstawowe funkcjonalności, takie jak planowanie zadań, zarządzanie pamięcią, odzyskiwanie błędów i interakcja z systemem pamięci masowej.
  - Biblioteki:
  - Spark SQL: Do przetwarzania danych strukturalnych przy użyciu zapytań podobnych do SQL.
  - Spark Streaming: Do przetwarzania strumieni danych w czasie rzeczywistym.
  - MLlib: Biblioteka do uczenia maszynowego.
  - GraphX: Do przetwarzania grafów.
  - PySpark: API Pythona dla Spark.
- 3. Przetwarzanie danych
  - Obliczenia w pamięci: Wykonuje operacje w pamięci, zmniejszając wejście/wyjście dysku i zwiększając szybkość.
  - Resilient Distributed Datasets (RDD): Niezmienne rozproszone zbiory danych.
  - DataFrames i Datasets: Interfejsy API wyższego poziomu do przetwarzania danych ustrukturyzowanych i półustrukturyzowanych.
  - Lazy Evaluation: Optymalizuje plany wykonania, odraczając obliczenia do momentu, gdy wynik będzie wymagany.
- 4. Obsługiwane źródła danych
  - Obsługuje integrację z:
  - HDFS (Hadoop Distributed File System)
  - Apache Cassandra
  - Apache HBase
  - Apache Hive
  - Amazon S3
  - JDBC/ODBC dla baz danych
  - Formaty JSON, CSV, Parquet, ORC, Avro

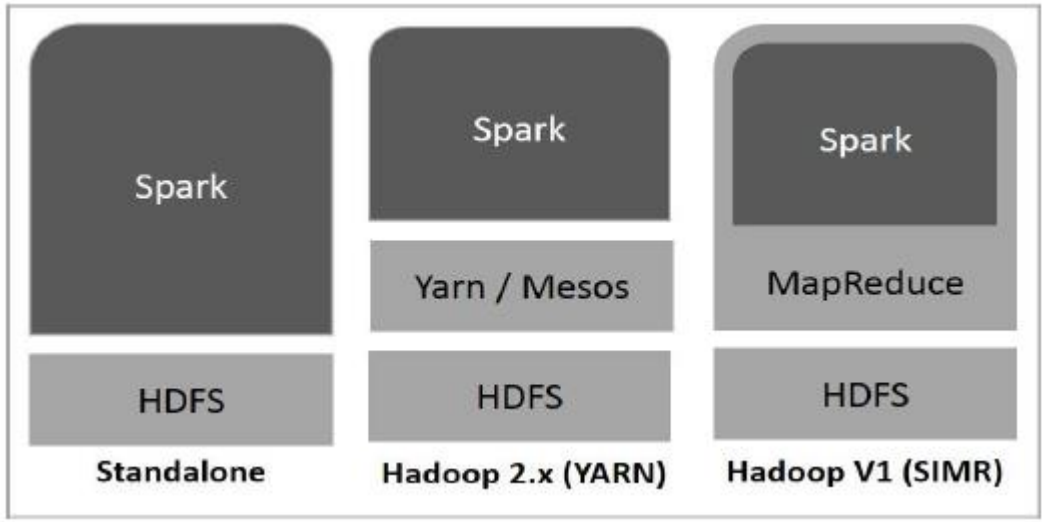
PODSTAWOWE CECHY APACHE SPARK

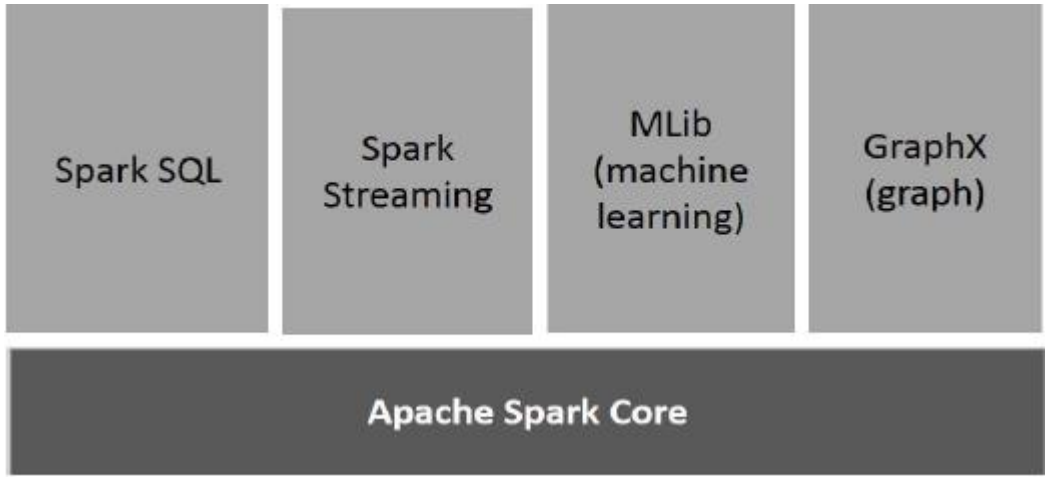
- 5. Optymalizacja wydajności
  - Catalyst Optimizer: Optymalizuje plany wykonania zapytań dla Spark SQL.
  - Tungsten Execution Engine: Optymalizuje fizyczne wykonanie dzięki ulepszonemu generowaniu kodu i zarządzaniu pamięcią.
- 6. Skalowalność i tolerancja błędów
  - Skalowanie poziome: Skalowanie w tysiącach węzłów. Tolerancja błędów: Osiągnięta dzięki informacjom o pochodzeniu w RDD, umożliwiając ponowne obliczenie utraconych danych.
- 7. Obsługa języków
  - Obsługuje wiele języków:
  - Scala (natywny)
  - Python (PySpark)
  - Java
  - R (SparkR)
- 8. Tryby wdrażania
  - Tryb autonomiczny: Działa jako niezależny menedżer klastra.
  - YARN: Zintegrowany z menedżerem zasobów Hadoop.
  - Mesos: Zgodny z Apache Mesos.
  - Kubernetes: Działa w środowiskach konteneryzowanych.
- 9. Przetwarzanie w czasie rzeczywistym
  - Strukturalne przesyłanie strumieniowe: Przetwarza strumienie danych w czasie rzeczywistym za pomocą deklaratywnego interfejsu API.

PODSTAWOWE CECHY APACHE SPARK

- 10. Interfejsy API i interfejsy
  - Funkcjonalne i obiektowe interfejsy API do pisania zadań transformacji i akcji.
  - Interfejs SQL do wykonywania zapytań dotyczących danych strukturalnych.
- 11. Zgodność z ekosystemem
  - Zgodność z różnymi narzędziami Big Data (np. Hadoop, Hive, Kafka, Flink).
  - Łatwa integracja z usługami w chmurze (np. AWS, Azure, GCP).
- 12. Zarządzanie zasobami
  - Dynamiczna alokacja: automatycznie dostosowuje alokację zasobów na podstawie obciążenia.
- 13. Oprogramowanie typu open source
  - Apache Spark jest oprogramowaniem typu open source, aktywnie rozwijanym i utrzymywanym w ramach Apache Software Foundation.
  - Ten bogaty zestaw funkcji sprawia, że Apache Spark jest niezwykle wszechstronny dla inżynierów danych, analityków i programistów pracujących z Big Data.

SPARK, HDFS I MODUŁY





SKRYPTY W JĘZYKACH SCALA, JAVA I PYTHON

SCALA – PODSTAWY

- Scalable Language
- Początki w roku 2001
- Wszystko jest obiektem
- Posiada cechy programowania funkcyjnego
- Język kompilowany

PODSTAWOWE TERMINY W JĘZYKU SCALA

- Obiekt: tabela, osoba, samochód
- Klasa: właściwości i zachowanie, projekt dla obiektu
- Metoda: zachowanie klasy, może być wiele metod w klasie
- Closure: funkcja działająca tylko w obrębie kodu, w którym została zdefiniowana
- Traits: są używane do definiowania typów obiektów poprzez specyfikację sygnatury wspieranych metod – analogicznie jak interfejs
w Javie

PRZYKŁADY (1/4)

- var Var1 = "Zmienna1"
- val Var2 : String = "Zmienna1"
- var Var4 = 2
- var Var5 = 3
- Var4+Var5
- Var4==Var5

PRZYKŁADY (2/4)

- var Var3 =1
- if (Var3 ==1) { println("Prawda") } else { println("Fałsz")}

- for(a <- 1 to 10)
- { println("Wartosc a: " + a); }

PRZYKŁADY (3/4)

- def funkcja(m: Int): Int = m * 10
- funkcja(2)

- var tablica1 = Array("A","B","C", "D", "E")
- var tablica2:Array[String] = new Array[String](3)
- var tablica3 = new Array[String](3)
- tablica3(0)="test"
- tablica1
- tablica1(2)

PRZYKŁADY (4/4)

- val lista1 = List(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
- val lista2 = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))

- lista1(0)

- object Program {
- def main(args: Array[String]) {
- println("Hello world!")
- }
- }

- Program.main(Array("A"))

PRZYKŁAD W APACHE SPARK

- val dane = Array(1, 2, 3, 4, 5,6,7,8,9,10)
- val daneRozproszone = sc.parallelize(dane)
- daneRozproszone.collect()

PRZYKŁAD 1. SCALA (1/2)

1. Sprawdź, czy jest dostęp do narzędzia scala.

```
scala --version
```

2. Uruchomić shell Apache Spark.

```
./bin/spark-shell
```

3. Załadować plik bigdata.txt.

```
val plik = sc.textFile("/home/solwit/dane/bigdata.txt")
```

4. Sprawdzić ile plik zawiera linii.

```
plik.count();
```

5. Sprawdzić zawartość pierwszej linii w pliku.

```
plik.first()
```

6. Utworzyć nowy podzbiór z wierszami zawierającymi słowo "data".

```
val wierszeInformatyki = plik.filter(wiersz => wiersz.contains("data"))
```

PRZYKŁAD 2. SCALA (2/2)

7. Sprawdzić ile linii ma podzbiór zawierający wiersze ze słowem data.

```
plik.filter(wiersz => wiersz.contains("data")).count()
```

8. Znaleźć najdłuższe słowo w pliku bigdata.txt.

```
plik.map(wiersz => wiersz.split(" ").size).reduce((a, b) => if (a > b) a else b)
```

9. Przeprowadzić działanie MapReduce na zbiorze danych.

```
val liczbaSlow = plik.flatMap(wiersz => wiersz.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a + b)
```

10. Wyświetlić kolekcję policzonych słów.

```
liczbaSlow.collect()
```

11. Przeładować zbiór wierszeInformatyki do pamięci RAM i sprawdzić ile linii jest w zbiorze.

```
wierszeInformatyki.cache()
wierszeInformatyki.count()
```

12. Wyświetlić zbiór wierszeInformatyki.

```
wierszeInformatyki.foreach(println)
```

13. Zapisać zbiór wierszeInformatyki jako plik tekstowy.

```
wierszeInformatyki.saveAsTextFile("wierszeInicjaly.txt")
```

14. Sprawdzić zapisany zbiór danych w systemie Linux.

PRZYKŁAD 2. SCALA

- scala> import org.apache.spark.SparkConf
- scala> val conf = new SparkConf().setAppName("test")
- scala> val dane=sc.textFile("/home/solwit/dane")
- scala> val tokeny = dane.flatMap(_.split(" "))
- scala> val liczbaSlow = tokeny.map((_, 1)).reduceByKey(_ + _)
- scala> liczbaSlow.sortBy(s => -s._2).map(x => (x._2, x._1)).top(10)

ZADANIA PODSUMOWUJĄCE - SCALA

1. Pobrać dowolną stronę internetową.
2. Wczytać zbiór danych w języku Scala.
3. Wykonać prostą analizę WordCount.

ZADANIA PODSUMOWUJĄCE – PYTHON

1. Napisać program, który za argument przyjmie liczbę i wypisze, czy liczba jest parzysta, czy też nie.

2. Napisać program, który wypisze wszystkie liczby parzyste z zakresu od 1 do 100.

3. Napisać funkcję, która będzie dodawała dwie liczby podane jako argument z zakresu od 1 do 100. W przypadku liczby spoza zakresu na wypisać błąd.

PRZYKŁADY PODSUMOWUJĄCE (1/2)

- 1. Sprawdź, czy jest dostęp do narzędzia Python w Apache Spark.
- ls ./bin/

- 2. Uruchomić shell Apache Spark.
- ./bin/pyspark

- 3. Załadować plik bigdata.txt.
- plik = sc.textFile("/home/solwit/dane/bigdata.txt")

- 4. Sprawdzić ile plik zawiera linii.
- plik.count()

- 5. Sprawdzić zawartość pierwszej linii w pliku.
- plik.first()

- 6. Utworzyć nowy podzbiór z wierszami zawierającymi słowo "data".
- wierszeInformatyki = plik.filter(lambda wiersz: "data" in wiersz)

- 7. Sprawdzić ile linii ma podzbiór zawierający wiersze ze słowem Informatyki.
- plik.filter(lambda wiersz: "data" in wiersz).count()

PRZYKŁADY PODSUMOWUJĄCE (2/2)

- 8. Znaleźć najdłuższe słowo w pliku bigdata.txt.
- plik.map(lambda wiersz: len(wiersz.split())).reduce(lambda a, b: a if (a > b) else b)

- 9. Przeprowadzić działanie MapReduce na zbiorze danych.
- liczbaSlow = plik.flatMap(lambda wiersz: wiersz.split()).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a+b)

- 10. Wyświetlić kolekcję policzonych słów.
- liczbaSlow.collect()

- 11. Przeładować zbiór wierszeInformatyki do pamięci RAM i sprawdzić ile linii jest w zbiorze.
- wierszeInformatyki.cache()
- wierszeInformatyki.count()

PRZYKŁAD W SCALA

1. scala> val plik = spark.read.textFile("/home/solwit/dane/bigdata.txt")
2. scala> plik.count()
3. scala> plik.first()
4. scala> val filtr = plik.filter(line => line.contains("data"))
5. scala> plik.filter(line => line.contains("data")).count()

OPERACJE NA DANYCH – MAPREDUCE W SCALI

1. scala> plik.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)
2. scala> import java.lang.Math
3. scala> plik.map(line => line.split(" ").size).reduce((a, b) => Math.max(a, b))
4. scala> val wordCounts = plik.flatMap(line => line.split(" ")).groupByKey(identity).count()
5. scala> wordCounts.collect()

PRZYKŁAD W PYSPARK

bin/pyspark

1. plik = spark.read.text("/home/solwit/dane/bigdata.txt")
2. plik.count()
3. plik.first()
4. filtr = plik.filter(plik.value.contains("data"))
5. plik.filter(plik.value.contains("data")).count()

OPERACJE NA DANYCH – MAPREDUCE W PYSPARK

```
1. from pyspark.sql.functions import *
2. plik.select(size(split(plik.value, "\s+")).name("numWords")).agg(max(col("numWords"))).collect()
3. wordCounts = plik.select(explode(split(plik.value, "\s+")).alias("word")).groupBy("word").count()
4. wordCounts.collect()
```

APLIKACJA W SCALA

- **import org.apache.spark.sql.SparkSession**
- **object SimpleApp** {
  - **def** main(args**:** Array[String]) {
    - **val** logFile **=** "bigdata.txt"
    - **val** spark **= SparkSession**.builder.appName("Aplikacja").getOrCreate()
    - **val** logData **=** spark.read.textFile(logFile).cache()
    - **val** numAs **=** logData.filter(line **=>** line.contains("a")).count()
    - **val** numBs **=** logData.filter(line **=>** line.contains("b")).count()
    - println(s"Wiersze z a: *$numAs*, Wiersze z b: *$numBs*")
    - spark.stop()
  - }
- }

APLIKACJA W PYTHON

- aplikacja.py
- from pyspark.sql import SparkSession
- logFile = "/home/solwit/dane/bigdata.txt"
- spark = SparkSession.builder.appName("Aplikacja").getOrCreate()
- logData = spark.read.text(logFile).cache()
- numAs = logData.filter(logData.value.contains('a')).count()
- numBs = logData.filter(logData.value.contains('b')).count()
- **print**("Lines with a: %*i*, lines with b: %*i*" % (numAs, numBs))
- spark.stop()

- ----
- spark-submit --master local[4] aplikacja.py

CACHE DLA DANYCH – SCALA VS. PYTHON

- linesWithSpark.cache()
- linesWithSpark.count()

SPARK SQL

SPARK SQL W SCALA

- val sqlcontext = new org.apache.spark.sql.SQLContext(sc)
- val dfs = sqlcontext.read.json("people.json")
- dfs.show()
- dfs.printSchema()
- dfs.select("name").show()
- dfs.filter(dfs("age") > 23).show()
- dfs.groupBy("age").count().show()

SPARK SQL

- from pyspark.sql import SparkSession
- spark1 = SparkSession.builder.appName("Aplikacja SparkSQL").config("spark.opcja", "0").getOrCreate()

TWORZENIE DATAFRAME

- df = spark.read.json("/home/solwit/dane/people.json")
- df.show()

OPERACJE NA DANYCH

- df.printSchema()
- df.select("name").show()
- df.select(df['name'], df['age'] + 1).show()
- df.filter(df['age'] > 21).show()
- df.groupBy("age").count().show()

PODSTAWOWE OPERACJE SQL

- df.createOrReplaceTempView("people")

- sqlDF = spark.sql("SELECT * FROM people")
- sqlDF.show()

GLOBALNA PERSPEKTYWA TYMCZASOWA

- df.createGlobalTempView("people")
- spark.sql("SELECT * FROM global_temp.people").show()
- spark.newSession().sql("SELECT * FROM global_temp.people").show()

SPARK STREAMING

SPARK STREAMING

PRZYKŁAD SPARK STREAMING (1/2)

- from pyspark import SparkContext
- from pyspark.streaming import StreamingContext

- # tworzenie lokalnego StreamingContext z dwoma wątkami i interwałem jednosekundowym
- sc = SparkContext("local[2]", "WordCount")
- ssc = StreamingContext(sc, 1)

PRZYKŁAD SPARK STREAMING (2/2)

```
# tworzenie połączenia Dstream, który będzie otrzymywany strumieniowo z serwera
lines = ssc.socketTextStream("localhost", 9999)

# Dzieli wiersz na wyrazy
words = lines.flatMap(lambda line: line.split(" "))

# liczy każdą parę wyrazów
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

# wyświetla pierwszych 10 wyrazów z RDD generowanego z DStream na konsolę
wordCounts.pprint()

ssc.start()             # rozpoczęcie obliczeń
ssc.awaitTermination()  # oczekiwanie na zakończenie

$ nc -lk 9999
```

SPARK SQL

SPARK SQL W SCALA

- val sqlcontext = new org.apache.spark.sql.SQLContext(sc)
- val dfs = sqlcontext.read.json("people.json")
- dfs.show()
- dfs.printSchema()
- dfs.select("name").show()
- dfs.filter(dfs("age") > 23).show()
- dfs.groupBy("age").count().show()

SPARK SQL

- from pyspark.sql import SparkSession
- spark1 = SparkSession.builder.appName("Aplikacja SparkSQL").config("spark.opcja", "0").getOrCreate()

TWORZENIE DATAFRAME

- df = spark.read.json("/home/solwit/dane/people.json")
- df.show()

OPERACJE NA DANYCH

- df.printSchema()
- df.select("name").show()
- df.select(df['name'], df['age'] + 1).show()
- df.filter(df['age'] > 21).show()
- df.groupBy("age").count().show()

PODSTAWOWE OPERACJE SQL

- df.createOrReplaceTempView("people")

- sqlDF = spark.sql("SELECT * FROM people")
- sqlDF.show()

GLOBALNA PERSPEKTYWA TYMCZASOWA

- df.createGlobalTempView("people")
- spark.sql("SELECT * FROM global_temp.people").show()
- spark.newSession().sql("SELECT * FROM global_temp.people").show()

SPARK STREAMING

SPARK STREAMING

PRZYKŁAD SPARK STREAMING (1/2)

- from pyspark import SparkContext
- from pyspark.streaming import StreamingContext

- # tworzenie lokalnego StreamingContext z dwoma wątkami i interwałem jednosekundowym
- sc = SparkContext("local[2]", "WordCount")
- ssc = StreamingContext(sc, 1)

PRZYKŁAD SPARK STREAMING (2/2)

```
# tworzenie połączenia Dstream, który będzie otrzymywany strumieniowo z serwera
lines = ssc.socketTextStream("localhost", 9999)

# Dzieli wiersz na wyrazy
words = lines.flatMap(lambda line: line.split(" "))

# liczy każdą parę wyrazów
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

# wyświetla pierwszych 10 wyrazów z RDD generowanego z DStream na konsolę
wordCounts.pprint()

ssc.start()             # rozpoczęcie obliczeń
ssc.awaitTermination()  # oczekiwanie na zakończenie

$ nc -lk 9999
```

ZADANIE

- 1. Napisać w Python program, który policzy liczbę wystąpień słowa scala. Wykorzystać dowolny plik z danymi.

- 2. Uruchomić program.

SPARK RDD

DWA TYPY OPERACJI

- Transformacje – tworzenie nowego RDD na podstawie istniejącego
- Akcje – obliczenia i agregacje zwracające wyniki

- Leniwe transformacje – kolejkowanie

PRZYKŁAD AKCJI W JAVIE

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
 public class PrintRDD {
    public static void main(String[] args) {
      // configure spark
      SparkConf sparkConf = new SparkConf().setAppName("Print Elements of RDD")
                                .setMaster("local[2]").set("spark.executor.memory","2g");
      // start a spark context
      JavaSparkContext sc = new JavaSparkContext(sparkConf);
      // read text files to RDD
      JavaRDD<String> lines = sc.textFile("data/rdd/input/file1.txt");
      // collect RDD for printing
      for(String line:lines.collect()){
          System.out.println("* "+line);
      }
    }
}
```

AKCJA W PYTHON

```
import sys
from pyspark import SparkContext, SparkConf
if __name__ == "__main__":
  # create Spark context with Spark configuration
  conf = SparkConf().setAppName("Print Contents of RDD - Python")
  sc = SparkContext(conf=conf)
  # read input text file to RDD
  rdd = sc.textFile("data/rdd/input/file1.txt")
  # collect the RDD to a list
  list_elements = rdd.collect()
  # print the list
  for element in list_elements:
    print(element)
```

AKCJA FOREACH

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.VoidFunction;

public class PrintRDD {

    public static void main(String[] args) {
        // configure spark
        SparkConf sparkConf = new SparkConf().setAppName("Print Elements of RDD")
                                .setMaster("local[2]").set("spark.executor.memory","2g");
        // start a spark context
        JavaSparkContext sc = new JavaSparkContext(sparkConf);

        // read text files to RDD
        JavaRDD<String> lines = sc.textFile("data/rdd/input/file1.txt");

        lines.foreach(new VoidFunction<String>(){
            public void call(String line) {
                System.out.println("* "+line);
        }});
    }
}
```

AKCJA FOREACH W PYTHON

```
import sys

from pyspark import SparkContext, SparkConf

if __name__ == "__main__":

  # create Spark context with Spark configuration
  conf = SparkConf().setAppName("Print Contents of RDD - Python")
  sc = SparkContext(conf=conf)

  # read input text file to RDD
  rdd = sc.textFile("data/rdd/input/file1.txt")

  def f(x): print(x)

  # apply f(x) for each element of rdd
  rdd.foreach(f)
```

ŁADOWANIE ZBIORU DANYCH

- **# sc to sparkcontext**
- **val data** = sc.textFile("train.tsv")

- data = sc.textFile("train.tsv")

WERYFIKACJA DANYCH

- data.getClass
- type(data)

TRANSFORMACJE

- **val reviews** = data.map(_.split("\t"))
- reviews = data.map(**lambda** x: x.split("\t"))

AKCJE

- reviews.count()

WORDCOUNT – PRZYKŁADY

- **val wordCounts** = reviews.flatMap(x => x(2).split(" ")).map((_, 1)).reduceByKey((a, b) => a + b).sortBy(_._2, false) wordCounts.take(5)

- word_counts = reviews.flatMap(**lambda** x : x[2].split()).map(**lambda** x: (x,1)).reduceByKey(**lambda** a,b: a + b).sortBy(**lambda** x: x[1],**False**)
- word_counts.take(5)

WYJAŚNIENIE DZIAŁANIA

- reviews.flatMap(x => x(2).split(" ")).take(5)
- word_counts = reviews.flatMap(**lambda** x : x[2].split()).take(5)

PONOWNA WERYFIKACJA

- reviews.flatMap(x => x(2).split(" ")).map((_, 1)).take(5)

- word_counts = reviews.flatMap(**lambda** x : x[2].split()).map(**lambda** x: (x,1)).take(5)
- word_counts.take(5)

KOLEJNA WERYFIKACJA

- reviews.flatMap(x => x(2).split(" ")).map((_, 1)).reduceByKey((a, b) => a + b).take(5)

- word_counts = reviews.flatMap(**lambda** x : x[2].split()).map(**lambda** x: (x,1)).reduceByKey(**lambda** a,b: a + b)
  word_counts.take(5)

NEGATYWNY WORDCOUNT

- **val negativeReviews** = reviews.filter(x(3).toInt == 0)
- **val negWordCounts** = negativeReviews.flatMap(x => x(2).split(" ")).map((_, 1)).reduceByKey((a, b) => a + b).sortBy(_._2, false) wordCounts.take(5)

NEGATYWNY WORDCOUNT

- negativeReviews = reviews.filter(**lambda** x: int(x[3]) == 0)
- word_counts = reviews.flatMap(**lambda** x : x[2].split()).map(**lambda** x: (x,1)).reduceByKey(**lambda** a,b: a + b).sortBy(**lambda** x: x[1],**False**)
- word_counts.take(5)

WYSZUKIWANIE FRAZY

- **val trueReviews** = reviews.filter(x => x(0) != "PhraseId")
- **val negativeReviews** = trueReviews.filter(x => x(3).toInt == 0)
- **val negWordCounts** = negativeReviews.flatMap(x => x(2).split(" ")).map((_, 1)).reduceByKey((a, b) => a + b).sortBy(_._2, false) negWordCounts.take(50)

PRZYKŁADY

- header = reviews.first()
- true_reviews = reviews.filter(**lambda** x: x != header)
- negativeReviews = true_reviews.filter(**lambda** x: int(x[3]) == 0)
- neg_word_counts = reviews.flatMap(**lambda** x : x[2].split()).map(**lambda** x: (x,1)).reduceByKey(**lambda** a,b: a + b).sortBy(**lambda** x: x[1],**False**)
- neg_word_counts.take(50)

POBIERANIE PLIKÓW JAKO DATAFRAME

- **val rawFlights** = sc.textFile("../Downloads/2008.csv")
- rawFlights.take(5)

- rawFlights = sc.textFile("../Downloads/2008.csv")
- rawFlights.take(5)

OPCJE POBIERANIA PLIKÓW

- **val df** = sqlContext.read.format("com.databricks.spark.csv").option("header", "true").load("../Downloads/2008.csv")
- df.take(5)

- **df** = sqlContext.read.format("com.databricks.spark.csv").option("header", "true").load("../Downloads/2008.csv")
- df.take(5)

PRZEGLĄD POBRANEGO SCHEMATU

- df.printSchema()

ZMIANA FORMATU

- df.col("Year").cast("int")

ZMIANA NAZW KOLUMN I TYPU DANYCH

- **val df_1** = df.withColumnRenamed("Year","oldYear")
- **val df_2** = df_1.withColumn("Year",df_1.col("oldYear").cast("int")).drop("oldYear")

- df_1 = df.withColumnRenamed("Year","oldYear")
- df_2 = df_1.withColumn("Year",df_1.col("oldYear").cast("int")).drop("oldYear")

KONWERSJA KOLUMN

```
def convertColumn(df: org.apache.spark.sql.DataFrame, name:String, newType:String) = {
  val df_1 = df.withColumnRenamed(name, "swap")
  df_1.withColumn(name, df_1.col("swap").cast(newType)).drop("swap")
}
```

KONWERSJA KOLUMN – DRUGI PRZYKŁAD

```
def convertColumn(df, name, new_type):
    df_1 = df.withColumnRenamed(name, "swap")
    return df_1.withColumn(name, df_1.col("swap").cast(new_type)).drop("swap")
```

WYKORZYSTANIE METODY

- val df_3 = convertColumn(df_2, "ArrDelay", "int")
- val df_4 = convertColumn(df_2, "DepDelay", "int")

- df_3 = convertColumn(df_2, "ArrDelay", "int")
- df_4 = convertColumn(df_2, "DepDelay", "int")

AGREGACJE DANYCH

- **val averageDelays** = df_4.groupBy(df_4.col("FlightNum")).agg(avg(df_4.col("ArrDelay")), avg(df_4.col("DepDelay")))

- averageDelays = df_4.groupBy(df_4.col("FlightNum")).agg(avg(df_4.col("ArrDelay")), avg(df_4.col("DepDelay")))

PRZYSPIESZANIE OBLICZEŃ I WYŚWIETLANIE

- averageDelays.cache()
- averageDelays.show()

SORTOWANIE WYNIKÓW

- averageDelays.orderBy("AVG(ArrDelay)").show() // ascending
- averageDelays.sort($"AVG(ArrDelay)".desc).show() // descending
- averageDelays.sort($"AVG(ArrDelay)".desc, $"AVG(DepDelay)".desc).show()

KONWERSJA W PYSPARK NA PANDAS DF

- df.toPandas()
- spark_df = sc.createDataFrame(pandas_df)

JEŻELI NIE DZIAŁA...

- from pyspark.sql.functions import col
- df = df.withColumn('year' , col('year').cast('int'))