Spark – *Structured Streaming*

Wprowadzenie

W ramach tego warsztatu skorzystamy z:

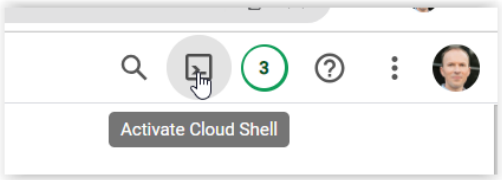
- Kafki jako źródła danych strumieniowych
- Narzędzie PySpark – Spark shell dla języka Python, w którym będziemy implementowali prototypy naszych programów przetwarzające strumienie danych.
- PostgreSQLa, do którego wprowadzimy wyniki naszych obliczeń

Dokładny format danych źródłowych oraz cel naszych obliczeń zostanie przedstawiony na późniejszym etapie.

Uruchomienie klastra

1. Przejdź do konsoli platformy GCP `https://console.cloud.google.com`

2. Korzystając z paska nawigacji (lewy górny róg konsoli), uruchom terminal *Cloud Shell*.



3. Uruchom polecenie, które utworzy klaster *Dataproc.* Klaster ten będzie zbudowany w oparciu o 3 maszyny wirtualne. Jedna z nich będzie zawierała komponenty typu „master", dwie pozostałe będą maszynami roboczymi. Oprogramowanie zainstalowane na tym klastrze będzie obejmowało (poza wieloma innymi narzędziami, których jednak nie skorzystamy)
    a. Klaster Hadoop
    b. Spark
    c. Klaster Kafki wraz z Zookeperem
    d. Platformę Docker dzięki której uruchomimy instancję bazy danych PostgreSQL
    e. Środowisko notatnikowe Jupyter

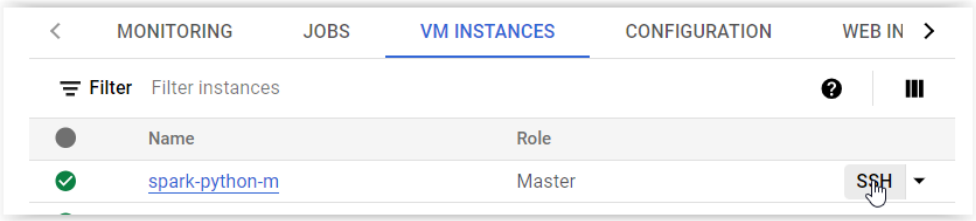
```
gcloud dataproc clusters create ${CLUSTER_NAME} \
--enable-component-gateway --region ${REGION} --subnet default \
--master-machine-type n1-standard-4 --master-boot-disk-size 50 \
--num-workers 2 --worker-machine-type n1-standard-2 --worker-boot-disk-size 50 \
--image-version 2.1-debian11 --optional-components JUPYTER,ZOOKEEPER,DOCKER \
--project ${PROJECT_ID} --max-age=3h \
--metadata "run-on-master=true" \
--initialization-actions \
gs://goog-dataproc-initialization-actions-${REGION}/kafka/kafka.sh
```

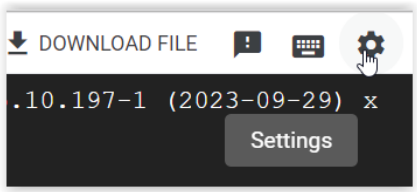
4. Polecenie zakończy swoje działanie w ciągu kilku minut. Ty jednak możesz kontynuować.

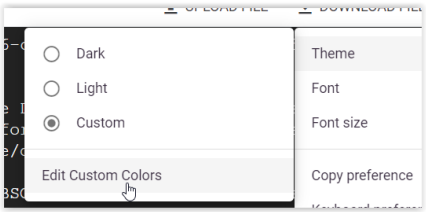
Przygotowania – uruchomienie terminali

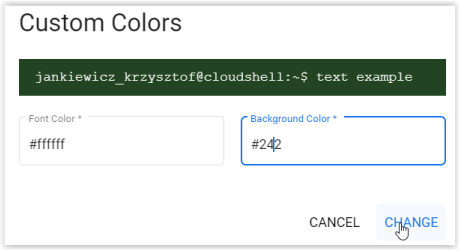
5. Uruchom trzy terminale SSH do maszyny master.
    a. **techniczny** – do operacji technicznych
    b. **nadawczy** – do tworzenia strumieni danych
    c. **odbiorczy** – do implementacji prototypów programów przetwarzające strumienie danych.

Możesz zmienić ich tło aby móc je rozróżniać.









Pierwszy test

Na początku sprawdzimy czy mechanizm *Structured Streaming* w ogóle działa.

6. W terminalu **nadawczym** wprowadź poniższe polecenie przygotowując się do „nadawania” na porcie 9876. **Zanotuj** nazwę węzła/klastra, na którego porcie serwera master będziemy działali.

```
CLUSTER_NAME=$(/usr/share/google/get_metadata_value attributes/dataproc-cluster-name)
echo ${CLUSTER_NAME}
nc -l ${CLUSTER_NAME}-m 9876
```

7. W terminalu **odbiorczym** uruchom PySpark, uruchamiając klaster Sparka na wątkach Javy (tryb local)

```
pyspark --master=local[2]
```

8. Utwórz, źródłowy `DataFrame` podłączając się do portu 9876.

```
import socket

host_name = socket.gethostname()

lines = spark \
    .readStream \
    .format("socket") \
    .option("host", host_name) \
    .option("port", 9876) \
    .load()
```

9. Dokonaj teraz kilku transformacji na źródłowym zbiorze danych (klasyka)

```
from pyspark.sql.functions import explode
from pyspark.sql.functions import split
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)

wordCounts = words.groupBy("word").count()
```

10. Czas na finał. Utwórz ujście naszego przetwarzania za pomocą metody `writeStream` określając typ wyniku, miejsce (na konsolę) i uruchamiając całą maszynerię.

```
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()
```

Pierwszy microbatch powinien się wykonać niezwłocznie.

```
-------------------------------------------
Batch: 0
-------------------------------------------
+----+-----+
|word|count|
+----+-----+
+----+-----+
```

11. W terminalu **nadawczym** wprowadź kilka linii z nadawanymi wartościami

```
spark scala scala spark structured streaming
spark scala scala spark
```

W terminalu **odbiorczym** powinno się rozpocząć przetwarzanie. Które ostatecznie wygeneruje wynik.

```
-------------------------------------------
Batch: 1
-------------------------------------------
+----------+-----+
|      word|count|
+----------+-----+
| streaming|    1|
|     scala|    4|
|     spark|    4|
|structured|    1|
|          |    1|
+----------+-----+
```

12. A teraz pytanie. Jaki wynik uzyskasz jeśli na wejściu wprowadzisz poniższą linię tekstu? Sprawdź czy poprawnie rozumiesz typ wyniku „complete”. W terminalu **nadawczym** wprowadź poniższą linię tekstu

```
spark scala scala spark
```

13. Po zakończeniu przetwarzania w terminalu **odbiorczym** zamknij obsługę obecnego ujścia (zapytania). Następnie wyjdź z narzędzia `spark-shell`

```
query.stop()
quit()
```

14. W terminalu **nadawczym** przerwij nadawanie za pomocą kombinacji klawiszy *Ctrl+C*.

KafkaSourceProvider – Data Source Provider dla Apache Kafka

Zabierzemy się teraz za bardziej ambitne zadanie. Będziemy przetwarzali dane z tematu (*topic*) systemu wymiany wiadomości *Apache Kafka*. Będą one tworzone za pomocą generatora przykładowych danych. Szczegóły za chwilę.

Przygotowanie środowiska

15. Na początku trochę przygotowań. W terminalu **technicznym** wyświetl dostępne obecnie tematy Kafki

```
CLUSTER_NAME=$(/usr/share/google/get_metadata_value attributes/dataproc-cluster-name)
kafka-topics.sh --list --bootstrap-server ${CLUSTER_NAME}-m:9092
```

16. Będziemy korzystali z tematu `kafka-input`. Jeśli temat `kafka-input` istnieje, to usuń go, a następnie utwórz go ponownie za pomocą poniższych poleceń.

```
kafka-topics.sh --create --bootstrap-server ${CLUSTER_NAME}-m:9092 \
    --replication-factor 2 \
    --partitions 3 --topic kafka-input
```

Szczegóły przetwarzania

Do naszego tematu będą wprowadzane dane w formacie JSON. Będą one zawierały następujące składowe

- `house (string)` – dom/drużyna, dla której dana osoba zdobyła punkty
- `character (string)` – osoba, która zdobyła punkty
- `score (int)` – liczba zdobytych punktów
- `ts (timestamp)` – moment, w którym punkty zostały zdobyte

Tabela wejściowa zatem będzie zatem „nieskończonym” zbiorem rekordów o powyższej strukturze.

Naszym zadaniem będzie utrzymywanie statystyk dotyczących każdej drużyny. Chcemy znać nazwy drużyn oraz dla każdej z nich chcemy wiedzieć ile razy dla tej drużyny zdobyto jakieś punkty, jaka jest suma zdobytych punktów, a także ile różnych osób zdobyło punkty dla tej drużyny. Innymi słowy wynikowa tabela ma mieć 4 kolumny:

- `house (string)` – dom/drużyna
- `how_many (long)` – ile razy zdobyto punkty dla tej drużyny
- `sum_score (long)` – suma zdobytych punktów
- `no_characters (long)` – liczba różnych osób, która zebrała punkty dla tej drużyny

Tabela źródłowa oczywiście nie jest w rzeczywistości nieskończona. A tabela wynikowa nie jest wyliczana za każdym razem od nowa. Implementacja zadań *Spark Structured Streaming* polega na przyrostowym utrzymaniu tabeli wynikowej na podstawie nowych danych napływających ze źródła.

17. Zastanów się nad kilkoma kwestiami
    a. Które z powyższych agregatów daje się dokładnie wyliczyć w sposób przyrostowy, a które, do dokładnych obliczeń, wymagają danych detalicznych („nieskończonej” tabeli wejściowej)?
    b. Ile krotek będzie miała maksymalnie wynikowa tabela?
    c. Jakie tryby obsługi wyniku są w tym przypadku w ogóle do rozważenia, oraz jaki mają one wpływ na wymagane cechy od wykorzystanego ujścia?

Wsadowy prototyp

18. Zaczniemy od wersji wsadowej naszego programu. Gdy będziemy wiedzieli, że potrafimy napisać program, który poprawnie przetwarza dane wejściowe, zamienimy go na wersję przetwarzającą strumień danych. W **odbiorczym** terminalu SSH pobierz wymagany sterownik

```
wget https://jdbc.postgresql.org/download/postgresql-42.6.0.jar
```

19. Następnie uruchom `spark-shell` za pomocą poniższego polecenia

```
pyspark --master=local[2] --driver-class-path postgresql-42.6.0.jar \
  --jars postgresql-42.6.0.jar
```

20. Utwórz zmienną odpowiadającą nazwie Twojego hosta.

```
import socket
host_name = socket.gethostname()
```

21. W pierwszym poleceniu utwórzmy sobie `DataFrame` ciągów znaków. Podobne dane będziemy mieli po odczytaniu wiadomości z tematu Kafki.

```
values = ["""{"house": "Ravenclaw", "character": "Myrtle Warren",
  "score": "1", "ts": "2022-08-11 16:34:11.633"}""",
  """{"house": "Thunderbird", "character": "Albus Dumbledore",
  "score": "4", "ts": "2022-08-11 16:33:50.74"}""",
  """{"house": "Slytherin", "character": "Bartemius Crouch, Sr.",
  "score": "4", "ts": "2022-08-11 16:34:16.9"}""",
  """{"house": "Gryffindor", "character": "Gregory Goyle",
  "score": "8", "ts": "2022-08-11 16:34:04.269"}""",
  """{"house": "Gryffindor", "character": "Astoria Greengrass",
  "score": "7", "ts": "2022-08-11 16:33:52.508"}""",
  """{"house": "Horned Serpent", "character": "Yaxley",
  "score": "4", "ts": "2022-08-11 16:33:48.829"}"""]
from pyspark.sql.types import StringType
valuesDF = spark.createDataFrame(values, StringType())
```

22. Sprawdźmy jaka jest budowa naszego obiektu `valuesDF`

```
valuesDF.printSchema()
```

```
>>> valuesDF.printSchema()
root
 |-- value: string (nullable = true)
```

23. Aby dokonać konwersji `DataFrame` ciągu znaków w formacie JSON na `DataFrame`, który ma postać strukturalną na początek przygotujemy odpowiednią definicję

```
from pyspark.sql.types import StructType, StructField, TimestampType
dataSchema = StructType([
    StructField("house", StringType(), True),
    StructField("character", StringType(), True),
    StructField("score", StringType(), True),
    StructField("ts", TimestampType(), True)
   ])
```

24. A teraz z niej skorzystajmy

```
from pyspark.sql.functions import from_json, col
dataDF = valuesDF.select(
                    from_json(col("value").cast(StringType()), dataSchema)
                    .alias("val")) \
                .select(col("val.house"),col("val.character"),
                        col("val.score").cast("int").alias("score"),col("val.ts"))
```

25. Teraz Twoja kolej. Utwórz końcowy efekt przetwarzania – obiekt `resultDF`.
    Pamiętaj o funkcjach, których wyniki nie są w stanie obliczane dokładnie w sposób przyrostowy.

```
???
```

Obok zaprezentowany jest efekt końcowy.

```
resultDF.printSchema()
resultDF.show()
```

```
>>> resultDF.printSchema()
root
 |-- house: string (nullable = true)
 |-- how_many: long (nullable = false)
 |-- sum_score: long (nullable = true)
 |-- no_characters: long (nullable = false)

>>> resultDF.show()
22/09/01 11:24:22 WARN org.apache.spark.sql.catalys
avior can be adjusted by setting 'spark.sql.debug.m
+--------------+--------+---------+-------------+
|         house|how_many|sum_score|no_characters|
+--------------+--------+---------+-------------+
|     Ravenclaw|       1|        1|            1|
|    Gryffindor|       2|       15|            2|
|     Slytherin|       1|        4|            1|
|Horned Serpent|       1|        4|            1|
|   Thunderbird|       1|        4|            1|
+--------------+--------+---------+-------------+
```

26. Przygotujemy teraz nasze ujście danych. W **technicznym** terminalu SSH uruchom następujące polecenie tworząc kontener z bazą danych PostgreSQL.

```
docker run --name postgresdb \
-p 8432:5432 \
-e POSTGRES_PASSWORD=mysecretpassword \
-d postgres
```

27. Zaczekaj około 10 sekund po utworzeniu kontenera. Następnie zaloguj się do bazy danych za pomocą poniższego polecenia.

```
export PGPASSWORD='mysecretpassword'
psql -h localhost -p 8432 -U postgres
```

28. Utwórz tabelę docelową.

```
create database streamoutput;

\c streamoutput

create table housestats (
    house character(30) primary key,
    how_many integer,
    sum_score integer,
    no_characters integer
);
```

29. Spróbujmy zatem zapisać nasz wynik do tak utworzonej bazy danych. W **odbiorczym** terminalu SSH wykonaj poniższe polecenie.

```
resultDF.write \
  .format("jdbc") \
  .option("url", f"jdbc:postgresql://{host_name}:8432/streamoutput") \
  .option("dbtable", "housestats") \
  .option("user", "postgres") \
  .option("password", "mysecretpassword") \
  .option("trucate", "true") \
  .mode("overwrite") \
  .save()
```

30. W **technicznym** terminalu SSH sprawdź czy dane są tam dostępne.

```
select * from housestats;
```

```
streamoutput=# select * from housestats;
     house      | how_many | sum_score | no_characters
----------------+----------+-----------+---------------
 Slytherin      |        1 |         4 |             1
 Ravenclaw      |        1 |         1 |             1
 Horned Serpent |        1 |         4 |             1
 Gryffindor     |        2 |        15 |             2
 Thunderbird    |        1 |         4 |             1
(5 rows)
```

Jeśli wszystko zadziałało bez zarzutu, to znaczy, że czas na wersję przetwarzającą strumień danych. Zanim jednak się za to zabierzemy polecenia, które nas doprowadziły do sukcesu. Przydadzą się.

```
import readline;
print('\n'.join([str(readline.get_history_item(i + 1)) \
for i in range(readline.get_current_history_length())]))
```

Wersja przetwarzająca strumień danych

31. Wersja strumieniowa naszego programu będzie odczytywała dane z tematu Kafki i zapisywała całą tabelę wynikową po jej aktualizacji do tabeli w PostgreSQL.
    W terminalu **odbiorczym** wyjdź ze narzędzia `pyspark` i uruchom go ponownie za pomocą poniższego polecenia.

```
pyspark --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.3.2 \
  --driver-class-path postgresql-42.6.0.jar \
  --jars postgresql-42.6.0.jar
```

32. Utwórz źródłowy `DataFrame`, który będzie pobierał dane z tematu Kafki `kafka-input`.

```
import socket
host_name = socket.gethostname()

ds1 = spark.readStream \
  .format("kafka") \
  .option("kafka.bootstrap.servers", f"{host_name}:9092") \
  .option("subscribe", "kafka-input") \
  .load()
```

33. Zapoznaj się ze schematem `ds1`, a następnie przekształć źródłowy `DataFrame`, w taki, który będzie zawierał wartość przekonwertowaną do ciągu znaków. Będzie łatwiej ją w ten sposób dalej przetwarzać

```
ds1.printSchema()

from pyspark.sql.functions import expr
valuesDF = ds1.select(expr("CAST(value AS STRING)").alias("value"))
```

34. Dalszy ciąg, do `resultDF`, znasz.

```
???
```

35. Sprawdź czy wszystko pasuje.

```
resultDF.printSchema()
```

```
>>> resultDF.printSchema()
root
 |-- house: string (nullable = true)
 |-- how_many: long (nullable = false)
 |-- sum_score: long (nullable = true)
 |-- no_characters: long (nullable = false)
```

36. Na początek zobaczymy czy nasz mechanizm przetwarzania strumieni danych działa. Zdefiniujemy ujście na konsolę

```
query = resultDF.writeStream \
  .outputMode("complete") \
  .format("console") \
  .start()
```

37. Poczekaj aż otrzymasz wynik z pierwszego batcha.

```
Batch: 0
-------------------------------------------
+-----+--------+---------+-------------+
|house|how_many|sum_score|no_characters|
+-----+--------+---------+-------------+
+-----+--------+---------+-------------+
```

38. Następnie w **nadawczym** terminalu SSH pobierz generator oraz wykorzystywaną przez niego bibliotekę, a następnie wywołaj nasz generator danych.

```
wget https://jankiewicz.pl/bigdata/bigdata-sp/KafkaFakerProducer.jar
wget https://repo1.maven.org/maven2/net/datafaker/datafaker/1.4.0/datafaker-1.4.0.jar

CLUSTER_NAME=$(/usr/share/google/get_metadata_value attributes/dataproc-cluster-name)

java -cp /usr/lib/kafka/libs/*:datafaker-1.4.0.jar:KafkaFakerProducer.jar \
      KafkaFakerProducer ${CLUSTER_NAME}-w-0:9092 kafka-input json 2 5
```

39. Jeśli nasz program wygenerował kolejną paczkę (lub dwie) wynikowych danych, to znaczy, że mechanizm przetwarzania strumieni danych działa i możemy zatrzymać bieżące przetwarzanie

```
query.stop()
```

40. Teraz możemy przekierować wynik do naszego docelowego ujścia. Niestety *Spark Structured Streaming* nie wspiera bezpośrednio ujść typu JDBC. Aby się o tym przekonać możesz spróbować wykonać poniższy kod.

```
resultDF.writeStream \
  .format("jdbc") \
  .option("url", f"jdbc:postgresql://{host_name}:8432/streamoutput") \
  .option("dbtable", "housestats") \
  .option("user", "postgres") \
  .option("password", "mysecretpassword") \
  .start()
```

```
  File "/usr/lib/spark/python/lib/py4j-0.10.9.5-src.zip/py4j/protocol.py", line 326, in get_return_value
py4j.protocol.Py4JJavaError: An error occurred while calling o138.start.
: java.lang.UnsupportedOperationException: Data source jdbc does not support streamed writing
```

41. Na szczęście możemy skorzystać z ujścia zdefiniowanego za pomocą metody `foreachBatch`.

```
from pyspark.sql import DataFrame
streamWriter = resultDF.writeStream.outputMode("complete").foreachBatch (
  lambda batchDF, batchId:
    batchDF.write
           .format("jdbc")
           .mode("overwrite")
           .option("url", f"jdbc:postgresql://{host_name}:8432/streamoutput")
           .option("dbtable", "housestats")
           .option("user", "postgres")
           .option("password", "mysecretpassword")
           .option("truncate", "true")
           .save()
)
```

42. To co? Trzy, cztery?

```
query = streamWriter.start()
```

43. Przetwarzanie czeka w cuglach. Nalejmy trochę paliwa. W terminalu **nadawczym**, na początek odrobinka. Dwa rekordy na sekundę przez 5 sekund.

```
java -cp /usr/lib/kafka/libs/*:datafaker-1.4.0.jar:KafkaFakerProducer.jar \
      KafkaFakerProducer ${CLUSTER_NAME}-w-0:9092 kafka-input json 2 5
```

44. Sprawdźmy w **technicznym** terminalu czy coś się zmieniło.

```
streamoutput=# select * from housestats;
     house      | how_many | sum_score | no_characters
----------------+----------+-----------+---------------
 Slytherin      |        2 |         9 |             2
 Ravenclaw      |        4 |        16 |             4
 Wampus         |        1 |         1 |             1
 Horned Serpent |        2 |        10 |             2
 Pukwudgie      |        3 |        10 |             3
(5 rows)
```

45. Jeśli tak nam dobrze idzie, to może teraz cała na przód? W terminalu **nadawczym**, nalejmy trochę więcej niż odrobinkę. 20 000 rekordów na sekundę przez 5 sekund.

```
java -cp /usr/lib/kafka/libs/*:datafaker-1.4.0.jar:KafkaFakerProducer.jar \
      KafkaFakerProducer ${CLUSTER_NAME}-w-0:9092 kafka-input json 20000 5
```

46. *Spark Structured Streaming* dość szybko powinien sobie z tym poradzić, wszak do tego został stworzony.

```
streamoutput=# select * from housestats;
     house      | how_many | sum_score | no_characters
----------------+----------+-----------+---------------
 Ravenclaw      |    12354 |     61334 |           213
 Gryffindor     |    12546 |     62938 |           213
 Horned Serpent |    12409 |     61985 |           213
 Slytherin      |    12552 |     62836 |           213
 Hufflepuff     |    12447 |     62072 |           213
 Pukwudgie      |    12610 |     63244 |           213
 Thunderbird    |    12542 |     62570 |           213
 Wampus         |    12552 |     62762 |           213
(8 rows)
```

47. W **odbiorczym** terminalu SSH zakończ działanie wszystkich aktywnych zapytań, a następnie wyjdź ze `pysparka`.

```
for q in spark.streams.active:
    q.stop()

quit()
```

48. W **technicznym** terminalu SSH wyjdź z psql

```
\q
```

49. Sprawdź ile wiadomości przeszło przez nasz wejściowy temat Kafki

```
kafka-run-class.sh kafka.tools.GetOffsetShell \
     --broker-list ${CLUSTER_NAME}-w-0:9092 \
     --topic kafka-input --time -1
```

```
jankiewicz_krzysztof@cluster-0999-m:~$ kafka-run-class.sh kafka.tools.GetOffsetShell \
>      --broker-list ${CLUSTER_NAME}-w-0:9092 \
>      --topic kafka-input --time -1
kafka-input:0:37350
kafka-input:1:25157
kafka-input:2:37517
```

To co udało nam się napisać, to oczywiście prototyp aplikacji. Jest on jednak kompletny i przetwarza dane z docelowego źródła zapisując wyniki w docelowym ujściu. Przeniesienie tego kodu do pełnowymiarowej aplikacji uruchamianej dla przykładu za pomocą `spark-submit`, jest już tylko kropką nad i.
Cała koncepcja, wszystkie elementy systemu, począwszy od definicji źródła, poprzez transformacje, a skończywszy na ostatecznej wersji ujścia są gotowe.