Spark Structured Streaming
podstawy

Krzysztof Jankiewicz

Plan

- Wprowadzenie
- Podstawy *Structured Streaming*
- Model przetwarzania
  - typ obsługi wyniku
  - obsługa znaczników czasowych zdarzeń
  - gwarancje *exactly-once*
- *Structured Streaming API*
  - źródła
  - transformacje
  - uruchamianie zapytań strumieniowych
- Obsługa znaczników czasowych zdarzeń i zdarzeń opóźnionych

Wprowadzenie

- *Structured Streaming* jest skalowalnym, odpornym na błędy silnikiem przetwarzania danych strumieniowych opartym o silnik **Spark SQL**.
- Powyższe oznacza, że do przetwarzania danych strumieniowych możemy wykorzystać
  - DataFrame/Dataset API
  - języki programowania: Scala, Java, Python i R
- Podobnie jak w przypadku Spark Streaming, mechanizmy wykorzystujące punkty kontrolne (*checkpoints*) oraz *Write Ahead Logs* pozwalają na implementację rozwiązań 24/7
- *Structured Streaming* pojawił się w wersji Sparka 2.0
- Od wersji Sparka 2.2 stał się podstawowym rozwiązaniem przetwarzania danych strumieniowych

Podstawy

- Aby była możliwość skorzystania z *Structured Streaming* musimy mieć dostęp do obiektu `org.apache.spark.sql.SparkSession`
- Konieczny jest import potrzebnych podczas przetwarzania klas
- Następnie za pomocą metody `readStream` obiektu `SparkSession` należy utworzyć `DataFrame`, który będzie reprezentował dane źródłowe
- Utworzony w ten sposób `DataFrame` można interpretować jako nieskończone źródło danych strumieniowych
- W kolejnym kroku, w sposób zależny od potrzeb, dokonuje się implementacji standardowych transformacji obiektu `DataFrame/Dataset`
- Ostatecznie należy uruchomić odbieranie danych ze strumienia za pomocą metody `writeStream` tworząc obiekt `StreamingQuery`, aby ostatecznie za pomocą metody `awaitTermination` na tym obiekcie ukończyć implementację przetwarzania

```
val spark = SparkSession.
  builder.
  appName("StructuredNetworkWordCount").
  getOrCreate()
```

```
import spark.implicits._
```

```
val lines = spark.readStream.
  format("socket").
  option("host", "127.0.0.1").
  option("port", 9876).
  load()
```

```
nc -lk 9876
```

```
val words = lines.as[String].flatMap(_.split(" "))

val wordCounts = words.groupBy("value").count()
```

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

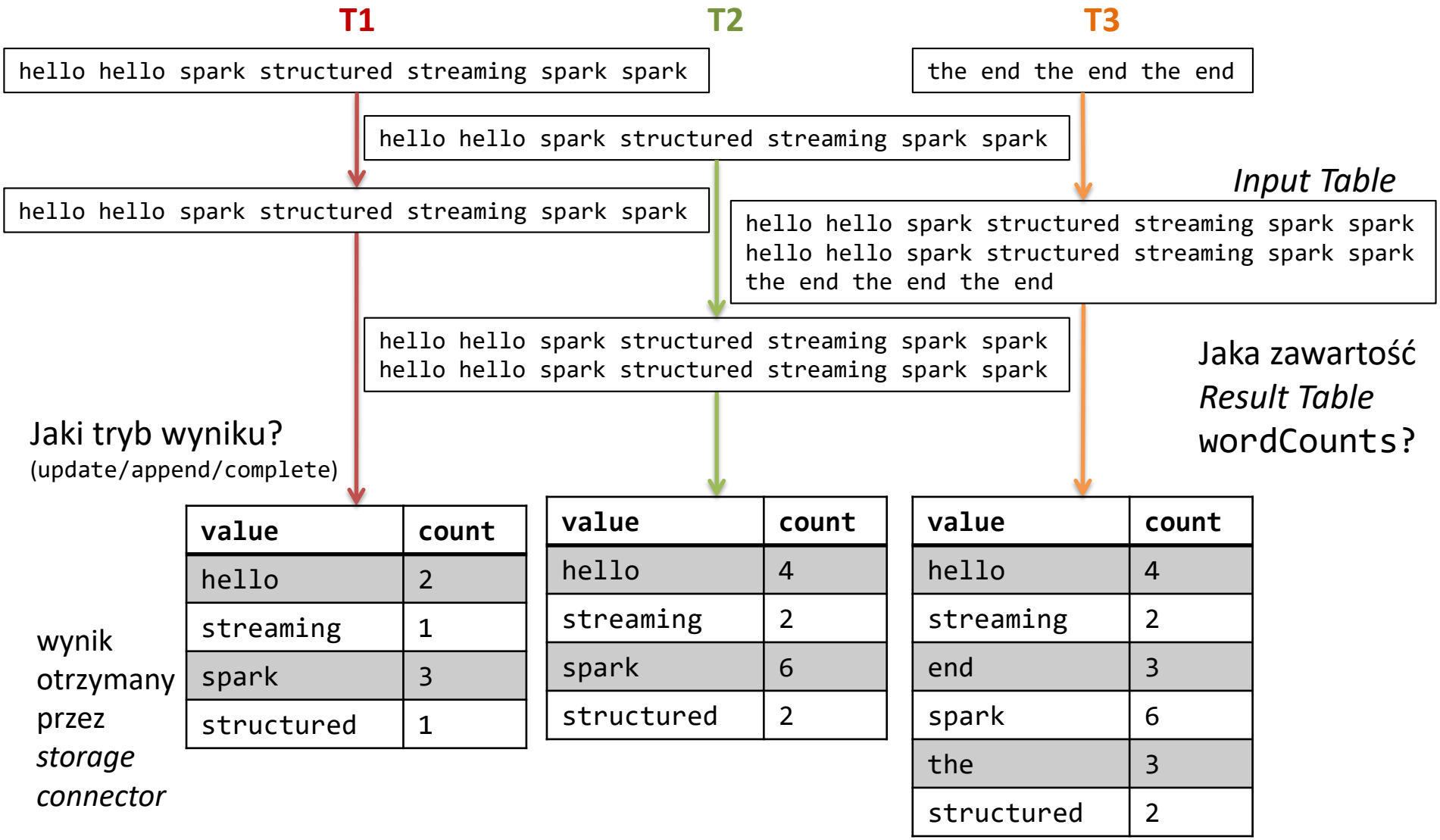
```
query.awaitTermination()
```

Model przetwarzania – tryb wyniku

- *Structured Streaming* (*StrStr*) traktuje źródłowy `DataFrame` jako **tabelę wejściową** (*Input Table*), która jest regularnie uzupełniana w oparciu o źródło danych
- Szereg transformacji na tabeli wejściowej definiuje **tabelę wynikową** (*Result Table*)
- Model przetwarzania pozwala programiście przetwarzanie strumieni danych analogicznie jak klasyczne wsadowe przetwarzanie, w wyniku którego:
  - Koncepcja: Spark przetwarza okresowo "**nieskończoną**" tabelę **wejściową**, tworząc nową postać tabeli wynikowej
  - Rzeczywistość: Spark na podstawie nowych danych otrzymywanych ze źródła utrzymuje tabelę **wynikową** w sposób **przyrostowy**

- Wyróżnia się trzy typy obsługi zmian tabeli wynikowej (trzy tryby generowania wyniku)
- Tryb wyniku (*outputMode*) definiuje co z tabeli wynikowej ostatecznie będzie "przepisywane" na wyjście (jaki będzie tworzony strumień danych wynikowych)
  - `complete` – **cała** tabela wynikowa – od złącza przechowywania (*storage connector*) będzie zależało jak będzie wyglądała ostateczna obsługa
  - `append` – jedynie **nowe** wiersze z tabeli wynikowej – zakłada się w tym wariancie, że istniejące wiersze tabeli wynikowej nie są modyfikowane
  - `update` – jedynie **zmodyfikowane** (w tym nowe) wiersze z tabeli wynikowej – jeśli transformacje nie zawierają agregacji, wówczas `update==append`

Model przetwarzania przykład (1/2)

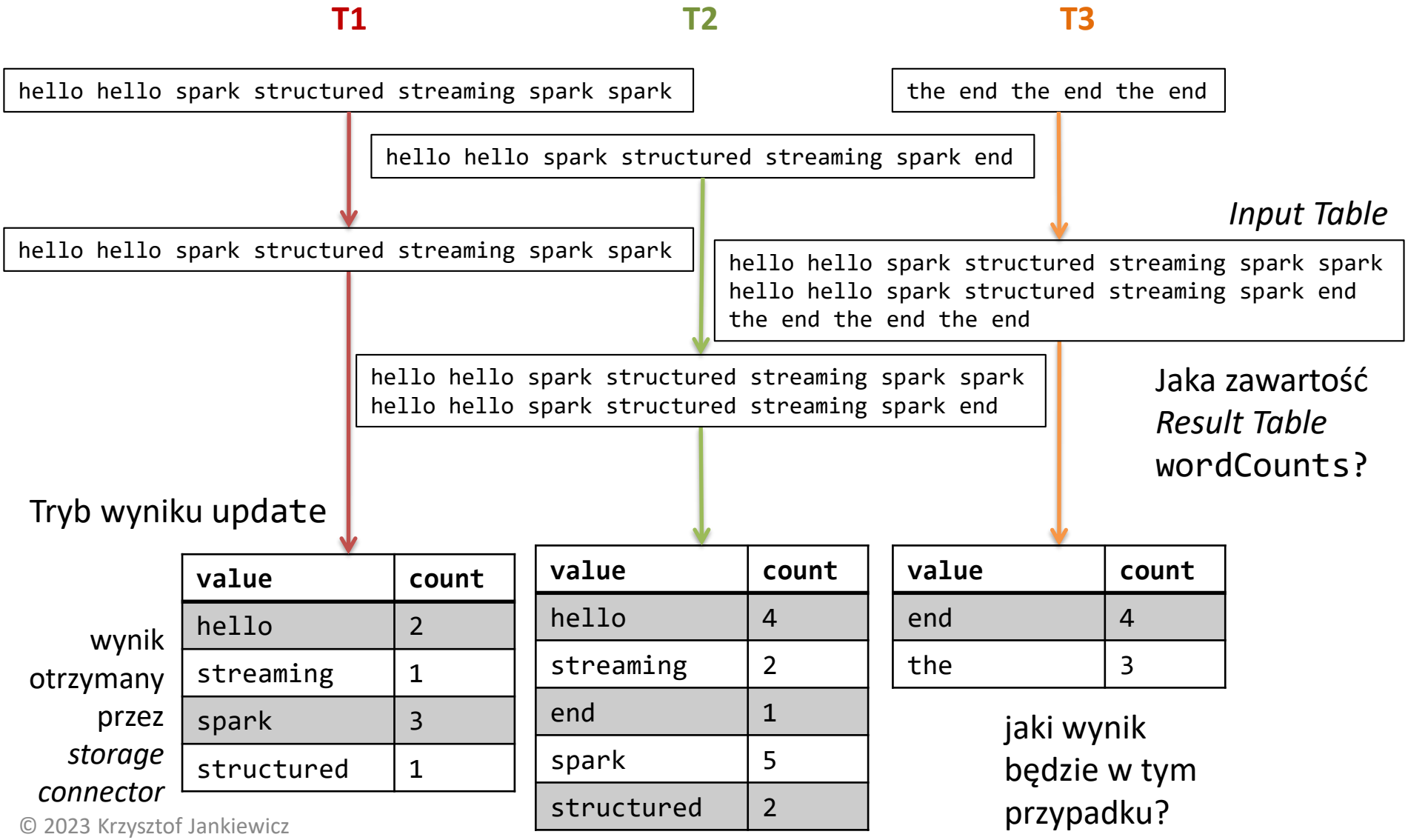


value	count
hello	2
streaming	1
spark	3
structured	1

value	count
hello	4
streaming	2
spark	6
structured	2

value	count
hello	4
streaming	2
end	3
spark	6
the	3
structured	2

Model przetwarzania przykład (2/2)



Model przetwarzania *event-time*, gwarancje *exactly-once*

- Z każdym zdarzeniem może zostać związany jego czas (*event-time*)
- Powiązanie zdarzenia z momentem jego wystąpienia odbywa się w systemach źródłowych
  (i nie jest związane z momentem pojawiania się zdarzenia w Sparku)
- Model przetwarzania *StrStr* obsługuje *event-time* dostarczając rozwiązań pozwalających na:
  - agregacje okna – np. zliczanie zdarzeń jakie miały miejsce w ciągu ostatniej minuty
  - obsługę zdarzeń opóźnionych – zdarzeń, których opóźnione przybycie należy wziąć po uwagę oraz zdarzeń, które spóźnione są tak bardzo, że nie powinny być uwzględniane w budowie tabeli wynikowej

- *StrStr* został zaprojektowany tak, aby dostarczać gwarancje przetwarzania *exactly-once*.
- Aby to osiągnąć został on zaprojektowany tak, aby obsłużyć każdy rodzaj awarii/błędu przez restart i/lub ponowne przetwarzanie.
- Rozwiązania, które temu służą to między innymi:
  - *offset* dla każdego źródła, aby można było śledzić przetworzone pozycje danych
  - punkty kontrolne oraz *write ahead logs*, które zapisują *offset* przetworzonych danych w ramach każdego wyzwalacza
  - idempotentna obsługa danych ze strumienia

Structured Streaming API
źródło

- Źródło – strumieniowy `DataFrame` można utworzyć korzystając z pośrednictwa interfejsu `DataStreamReader` (`SparkSession.readStream()`)
- W wersji Sparka 3.2 dostępne są trzy wbudowane źródła
  - **Pliki** – odporne na błędy, domyślnie wymaga podania schematu, opcje:
    - `path` – ścieżka której zawartość będzie monitorowana
    - `maxFilesPerTrigger` – maksymalna liczba plików obsługiwana przez wyzwalacz
    - `latestFirst` – przetwarzanie będzie się odbywało od najnowszych plików
    - `fileNameOnly` – pliki o tej samej nazwie, ale różnym miejscu występowania będą traktowane jako te same
  - **Gniazda TCP** – przeznaczony jedynie do testowania, brak odporności na błędy, opcje: `host`, `port`
  - **Rate source** – do testów, opcje: `rowsPerSecond`, `rampUpTime`, `numPartitions`
  - **Tematy Kafki** – odporne na błędy, najważniejsze opcje:
    - `kafka.bootstrap.servers` – serwery Kafki
    - `subscribe` – tematy Kafki

```
val lines = spark.readStream.
  format("socket").
  option("host", "127.0.0.1").
  option("port", 9876).
  load()
```

```
val tripSchema = new StructType().
  add("trip_id", "integer").
  add("starttime", "string").
  add("stoptime", "string").
  add("bikeid", "string").
  add("tripduration", "float").
  add("from_station_name", "string").
  add("to_station_name", "string").
  add("from_station_id", "string").
  add("to_station_id", "string").
  add("usertype", "string").
  add("gender", "string").
  add("birthyear", "integer")
```

```
val trips = spark.readStream.
  option("sep", ",").
  schema(tripSchema).
  csv("file:///home/username/trip")
```

Structured Streaming API
źródło – schemat wejściowej tabeli

- W zależności od typu źródła, różny jest schemat wejściowej tabeli (`DataFrame`)
  - Pliki – domyślnie wymaga podania schematu
  - Gniazdo TCP
  - Kafka
  - Rate source

```
[ value: string ]
```

```
[ key:       binary,
  value:     binary,
  topic:     string,
  partition: int,
  offset:    long,
  timestamp: long,
  timestampType: int ]
```

```
scala> val lines = spark.read$tream.
| format("socket").
| option("host", "127.0.0.1").
| option("port", 9876).
| load()
(. . .)
lines: org.apache.spark.sql.
DataFrame = [value: string]
```

```
scala> val dsl = spark.
| readStream.
| format("kafka").
| option("kafka.bootstrap.servers", "localhost:6667").
| option("subscribe", "kafka–to–ss").
| load()
dsl: org.apache.spark.sql.DataFrame = [key: binary,
value: binary ... 5 more fields]
```

```
scala> val trips = spark.readStream.
| option("sep", ",").
| schema(tripSchema).
| csv("file:///home/username/trip")
trips: org.apache.spark.sql.DataFrame =
[trip_id: int, starttime: string ... 10 more fields]
```

```
scala> val rates = spark.readStream.
  format("rate").
  option("rowsPerSecond", 1).
  load()

rates: org.apache.spark.sql.DataFrame =
[timestamp: timestamp, value: bigint]
```

Structured Streaming API transformacje proste

- Duża część metod dostępnych w klasycznych (wsadowych) `DataFrame/Dataset` jest także dostępna w ich wersjach strumieniowych
- Dostępne operacje **proste** (selekcja, projekcja, agregacja) obejmują:
  - nietypowane transformacje "*SQL-owe*" (np.: `select`, `where`, `groupBy`)
  - typowane transformacje "*RDD*" (np.: `map`, `filter`, `flatMap`, `groupByKey`)

```
import org.apache.spark.sql.functions.to_timestamp

val ttrips =  trips.select(to_timestamp($"starttime", "MM/dd/yyyy HH:mm") as "starttime",
  $"gender", $"tripduration")
val result = ttrips.groupBy("gender").
                    agg(sum("tripduration").as("sumtripdur"))
val query = result.writeStream.outputMode("update").
                   format("console").start()
```

```
case class Trip(starttime: java.sql.Timestamp, gender: String, tripduration: Float);

val ttripsds = ttrips.as[Trip]
val ttripsds2 = ttripsds.filter(t => (t.gender == "Male") ||
                                     (t.gender == "Female"))
val result = ttripsds2.groupByKey(t => t.gender).
                    agg(sum($"tripduration").as[Double])
val query = result.writeStream.outputMode("update").
                   format("console").start()
```

Structured Streaming API transformacje złożone (2.2)

- Oprócz prostych transformacji obejmujących selekcję, projekcję i agregację *StrStr* udostępniał w wersji 2.2 także:
  - **łączenie** strumieniowych zbiorów `DataFrames` ze statycznymi zbiorami `DataFrames`

```
trips.join(stationsDF, trips("from_station_id") == stationsDF("station_id"))
```

  - **eliminowanie duplikatów**

```
trips.dropDuplicates("trip_id")
```

```
trips.dropDuplicates("starttime")
```

  - metody `mapGroupsWithState` oraz `flatMapGroupsWithState` – dla bardziej złożonych operacji stanowych niż podstawowe agregacje
- Dlaczego nie wszystkie operacje dostępne dla statycznych `DataFrames` są dostępne dla ich odpowiedników strumieniowych?
  Wynika to z charakteru przetwarzania. Spark oblicza wynikową tabelę na podstawie zdefiniowanych transformacji w sposób przyrostowy
- Z wersji na wersje pojawiają się kolejne możliwości takie jak łączenie strumieni

Patrz:
`https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.streaming.GroupState`

Structured Streaming API transformacje niewspierane

- Wielopoziomowe agregacje (łańcuchy agregacji)
- Ograniczanie wyników do pierwszych N wierszy (`limit`).
- Eliminacje duplikatów za pomocą operacji `distinct`
- Sortowanie – dostępne jest jedynie po dokonanej agregacji w typie wyniku `complete`.
- Niektóre typy połączeń zewnętrznych pomiędzy zbiorem strumieniowym i statycznym
  - Full outer join
  - Left outer join ze strumieniowym Datasetem po prawej stronie
  - Right outer join ze strumieniowym Datasetem po lewej stronie
- Niektóre metody w pewnych przypadkach:
  - `count()` – niedostępne bezpośrednio na typie strumieniowym. Alternatywą jest wykorzystanie wyrażenia `ds.groupBy().count()`
  - `foreach()` – dostępne jest jedynie na wyniku `ds.writeStream.foreach(...)`.
  - `show()` – zamiast tego dostępne jest ujście wyniku na konsolę.

- Każdorazowo użycie powyższych operacji generuje wyjątek `AnalysisException`.

Structured Streaming API uruchamianie – podstawy

- Samo definiowanie transformacji na typie strumieniowym nie uruchamia przetwarzania (leniwe transformacje)
- Odpowiednikiem akcji, który kończy przetwarzanie jest
  - zdefiniowanie ujścia (*sink*)
  - uruchomienie (ciągłego) zadania za pomocą metody `start`
- Rozpoczyna się to od zdefiniowania obiektu `DataStreamWriter` zwracanego za pomocą metody `Dataset.writeStream()`
- Definiując obiekt `DataStreamWriter` określa się:
  - szczegóły takie jak format danych (np.: ORC, Parquet) i lokalizację
  - tryb generowania wyniku (`update`, `complete`, append)
  - nazwę zapytania w celu jego późniejszej identyfikacji (np. podczas monitorowania)
  - częstotliwość uruchamiania wyzwalacza – jeśli nie zostanie wyspecyfikowana, system sprawdzi dostępność nowych danych zaraz po zakończeniu wcześniejszej tury (*batch*) obliczeń
  - miejsce przechowywania punktów kontrolnych

Structured Streaming API
wyzwalacze – własności przetwarzania

- Standardowo StrStr implementuje podejście do przetwarzania danych strumieniowych *micro-batch*
- Gdzie jest *interval-batch* znany ze Spark Streaming?
- Częstotliwość uruchamiania przetwarzania zbuforowanych danych wejściowych określa tzw. *wyzwalacz*
- Wyzwalacze
  - definiuje się na ujściu, na obiekcie `DataStreamWriter`
  - domyślnie uruchamiają przetwarzanie natychmiast po zakończeniu poprzednich obliczeń (o ile jakieś dane czekają na wejściu)
  - mogą być uruchamiane co pewien okres czasu

```
trigger(Trigger.
  ProcessingTime("60 seconds"))
```

  - mogą uruchamiać przetwarzanie raz, a następnie zamykać zapytanie

```
trigger(Trigger.Once())
```

  - mogą implementować (eksperymentalnie) tzw. ciągłe przetwarzanie strumieniowe, które uruchamiane jest dla każdej nowej danej wejściowej)

```
trigger(Trigger.Continuous("1 second"))
```

tu czas określa częstotliwość tworzenia punktów kontrolnych

17

Structured Streaming API
uruchamianie – typy ujść – własności

Spark udostępnia szereg typów ujść (*sink*), każde z nich posiada swoje własności dotyczące: obsługiwanych typów wyników, poziomu odporności na błędy, wymaganych opcji

Typ ujścia	**Wspierane typy wyników**	**Opcje**	**Odporność na błędy (gwarancje)**
File	`append`	`path` – ścieżka w której wynikowe pliki będą zapisywane format pliku – opcje zależne od formatu	*exactly-once*
Kafka	`wszystkie`		*at-least-once*
Foreach	`wszystkie`		*at-least-once*
ForeachBatch	`wszystkie`		Zależne od implementacji funkcji
Console	`wszystkie`	`numRows` – liczba wierszy wyświetlana dla każdego wyzwalacza (domyślnie 20) `truncate` – czy duży wynik ma być skracany (domyślnie `true`)	nie
Memory	`append`, `complete`		nie, jedynie dla typu `complete` zrestartowane zapytanie ponownie utworzy kompletną tabelę

Structured Streaming API
ujścia, uruchamianie – przykłady

```
val query = result.writeStream.
  format("parquet").
  option("path", "realtime/trips").
  start()
```

```
val query = result.writeStream.
  trigger(ProcessingTime("60 seconds")).
  format("org.apache.spark.sql.execution.datasources.orc").
  option("checkpointLocation", checkpointDirectory).
  option("path", outputDirectory).
  start()
```

```
val query = result.writeStream.
  outputMode("update").
  format("console").
  queryName("first").
  start()
```

```
val query = result.writeStream.
  format("kafka").
  option("kafka.bootstrap.servers", " 127.0.0.1:6667").
  option("topic", "gender_trips").
  start()
```

```
val query = result.writeStream.
  foreach(...).
  start()
```

```
val query = result.writeStream.
  format("memory").
  queryName("gender_trips").
  start()
```

Structured Streaming API
ujścia, uruchamianie – forEach... przykłady

```
val query = result.writeStream.
  foreach(
    new ForeachWriter[String] {

      def open(partitionId: Long, version: Long): Boolean = {
        // Otwieramy połączenie np. z bazą danych
      }

      def process(record: String): Unit = {
        // Zapisujemy dane do ujścia
      }

      def close(errorOrNull: Throwable): Unit = {
        // Zamykamy połączenie
      }
  }).
  start()
```

```
val query = result.writeStream.foreachBatch {
  (batchDF: DataFrame, batchId: Long) =>
  // transformacje i zapis do ujścia batchDF
}.start()
```

Obsługa znaczników czasowych zdarzeń

- Agregacje na oknie przesuwnym wymagają
  - transformacji `groupBy`
  - funkcji `window()`
- Funkcja `window`
  - wskazuje atrybut danych traktowany jako znacznik czasu zdarzenia
  - określa długość okna
  - wyznacza częstotliwość jego ewaluacji

```
case class AccessLogRecord (
    clientIpAddress: String,
    rfc1413ClientIdentity: String,
    remoteUser: String,
    dateTime: String,
    request: String,
    httpStatusCode: String,
    bytesSent: String
)
```

```
val logRecords = ds2.flatMap(x => parser.parseRecord(x._2))
logRecords: org.apache.spark.sql.Dataset[AccessLogRecord] . . .

import org.apache.spark.sql.functions.to_timestamp

val logRecordsWTS = logRecords.select(
  to_timestamp($"dateTime", "dd/MMM/yyyy:HH:mm:ss").as("ts"),
  $"clientIpAddress", $"bytesSent")

val windLogRecords = logRecordsWTS.
  groupBy(window($"ts", "10 minutes", "5 minutes"),
          $"clientIpAddress").
  agg("bytesSent" -> "sum", "bytesSent" -> "count")
windLogRecords: org.apache.spark.sql.DataFrame =
  [window: struct<start: timestamp, end: timestamp>,
   clientIpAddress: string ... 2 more fields]
```

Obsługa danych opóźnionych i *watermark*

- Procesor danych strumieniowych korzystający z czasu zdarzeń powinien posiadać mechanizm pozwalający na obsługę zdarzeń nieuporządkowanych.
- W przypadku *StrStr* takim mechanizmem są znaczniki *watermarks*
- Wskaźniki *watermarks* pozwalają na obsługę danych opóźnionych (nieuporządkowanych)
- Dodanie do strumienia znaczników *watermarks* odbywa się za pomocą metody `withWatermark`
- Metoda `withWatermark` posiada dwa argumenty
  - wskazanie na atrybut danych zawierających znacznik czasu *`ets`* (zdarzenia), na podstawie którego będzie wyznaczany znacznik *watermark*
  - interwał *`wInt`* jaki będzie odejmowany od znacznika czasu w celu wyznaczenia znacznika *watermark*
  - znacznik *watermark* *`wm`* wyliczany jest zatem jako: *`wm = max(ets) - wInt`*
- Dane ze strumienia będą akceptowane (będą co najwyżej opóźnione) tylko wówczas, gdy ich znacznik czasowy *`ts`* spełni warunek *`ts >= wm`*

```
val windLogRecords = logRecordsWTS.
  withWatermark("ts", "10 minutes").
  groupBy(
    window($"ts", "10 minutes", "5 minutes"),
    $"clientIpAddress").
  agg("bytesSent" -> "sum", "bytesSent" -> "count")
```

dane strumienia, których znacznik czasowy *ts < wm* mogą zostać zignorowane jako spóźnione (*too late events*)

Czas – dwie semantyki

- Warto zwrócić uwagę, że w przypadku StrStr pojęcie czasu
  - ma dwie niezależne semantyki
    - semantyka zapytania
    - szczegóły dotyczące przetwarzania
  - operuje na różnych typach znaczników
    - zdarzeń
    - przetwarzania

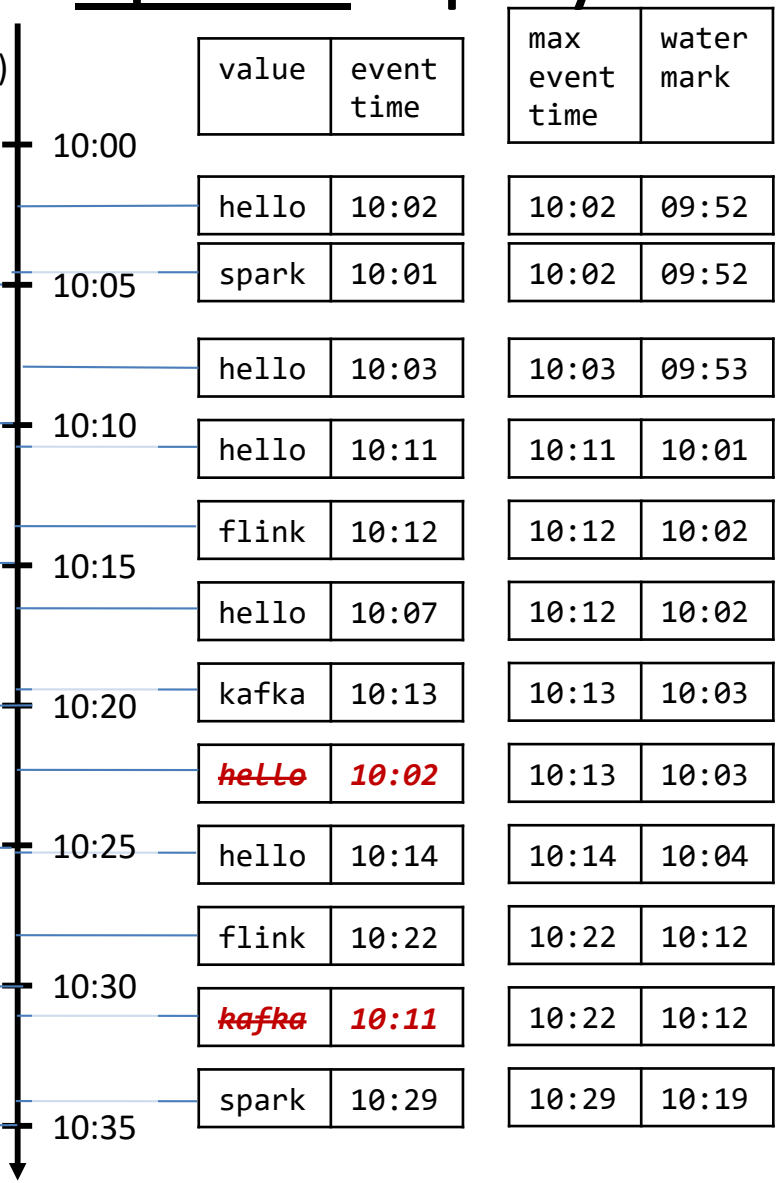
- Typy znaczników
  - zdarzeń – **definicja okna**, ***opóźnienia***
  - przetwarzania – **częstotliwość dostarczania wyników**, **definicja okna**, ***opóźnienia***

- Semantyka zapytania
  - **definicja okna** – wynikająca z logiki biznesowej
- Szczegóły dotyczące przetwarzania
  - ***obsługa danych opóźnionych*** – charakterystyka strumienia, tendencje do nieuporządkowania danych
  - **częstotliwość dostarczania wyników**

```
val windLogRecords = logRecordsWTS.
  withWatermark("ts", "10 minutes").
  groupBy(
    window($"ts", "10 minutes"),
    $"clientIpAddress").
  agg(sum($"bytesSent"),
      count($"bytesSent")).
  writeStream.
  trigger(Trigger.
          ProcessingTime("60 seconds")).
  start()
```

Obsługa danych opóźnionych
tryb update – przykład

W założeniu:
długość okna – 10 minut
przesunięcie – 5 minut
watermark – 10 minut
trigger – *ProcessingTime("5 minutes")*
dane dostarczane są z opóźnieniem
event-time <> processing time



window	value	count
09:55-10:05	hello	1
09:55-10:05	spark	1
10:00-10:10	hello	1
10:00-10:10	spark	1

window	value	count
09:55-10:05	hello	2
10:00-10:10	hello	2

window	value	count
10:00-10:10	hello	3
10:05-10:15	hello	2
10:05-10:15	kafka	1
10:10-10:20	kafka	1

?

?

?

?

- Czym są powyższe tabele?
  Tabelą wynikową, czy zdarzeniami kierowanymi do ujścia?
- Co kryje się pod znakami zapytania?
- Jak wygląda tabela wynikowa w pierwszych trzech przypadkach?

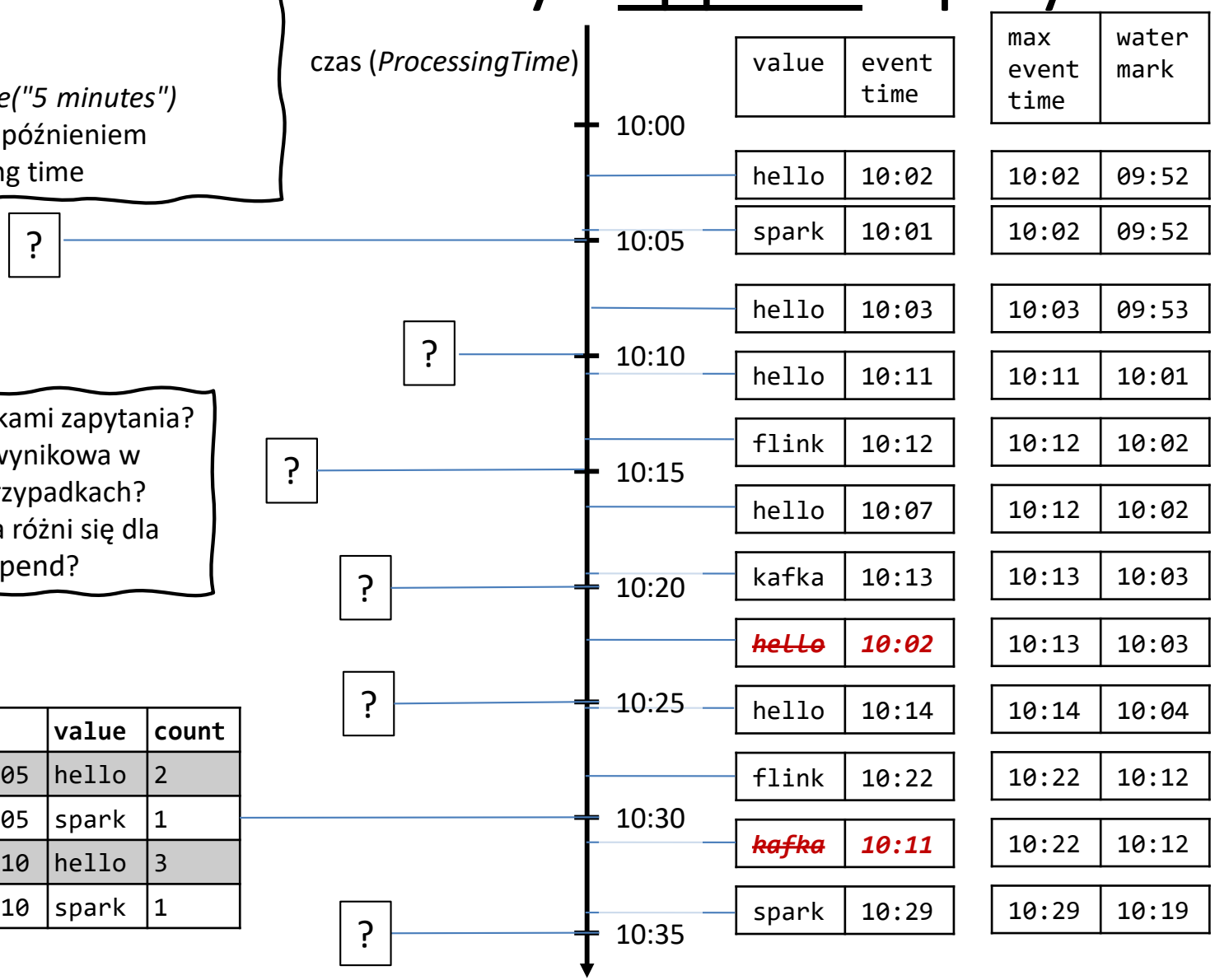
Obsługa danych opóźnionych tryb append

- Obsługa danych opóźnionych w trybie `update` wymaga funkcjonalności ujścia pozwalającej na
  - drobnoziarnistą aktualizację
    np.: `09:55-10:05`, `hello`, `1` => `09:55-10:05`, `hello`, **`3`**
  - lub kompensację nadmiarowych informacji, dostarczającą najnowszych wersji danych
- Dlatego też czasami konieczne jest wykorzystanie trybu `append`, który daje tylko ostateczne wyniki – takie, których nie będzie trzeba już zmieniać
- Wykorzystanie trybu `append` determinuje opóźnienie w wysyłaniu danych

Obsługa danych opóźnionych
tryb append – przykład

W założeniu:
długość okna – 10 minut
przesunięcie – 5 minut
watermark – 10 minut
trigger – *ProcessingTime("5 minutes")*
dane dostarczane są z opóźnieniem
event-time <> processing time

- Co kryje się pod znakami zapytania?
- Jak wygląda tabela wynikowa w pierwszych trzech przypadkach?
- Czy tabela wynikowa różni się dla trybów `update` i append?



window	value	count
09:55-10:05	hello	2
09:55-10:05	spark	1
10:00-10:10	hello	3
10:00-10:10	spark	1

Obsługa danych opóźnionych

Tolerancja dla danych opóźnionych	Mniejsza	Większa
Wielkość przetwarzanych danych	Mniejsza	Większa
Potrzeby pamięciowe	Mniejsze	Większe
Opóźnienia w wysyłaniu danych (tryb append)	Mniejsze	Większe

Podsumowanie – od źródła do ujścia

```
val rates = spark.
  readStream.
  format("rate").
  option("rowsPerSecond", 1000000).
  load().
  withColumn("modulo", $"value" % 10).
  groupBy($"modulo").
  agg(sum($"value").as("sumval")).
  writeStream.
  format("memory").
  queryName("modulo_query").
  trigger("1 minute").
  outputMode("update").
  option("checkpointLocation",
         "/tmp/_checkpoints/modulo").
  start()
```

Rate DataFrame

timestamp	value
1647192663	1
1647192683	2
1647192691	3
…	…

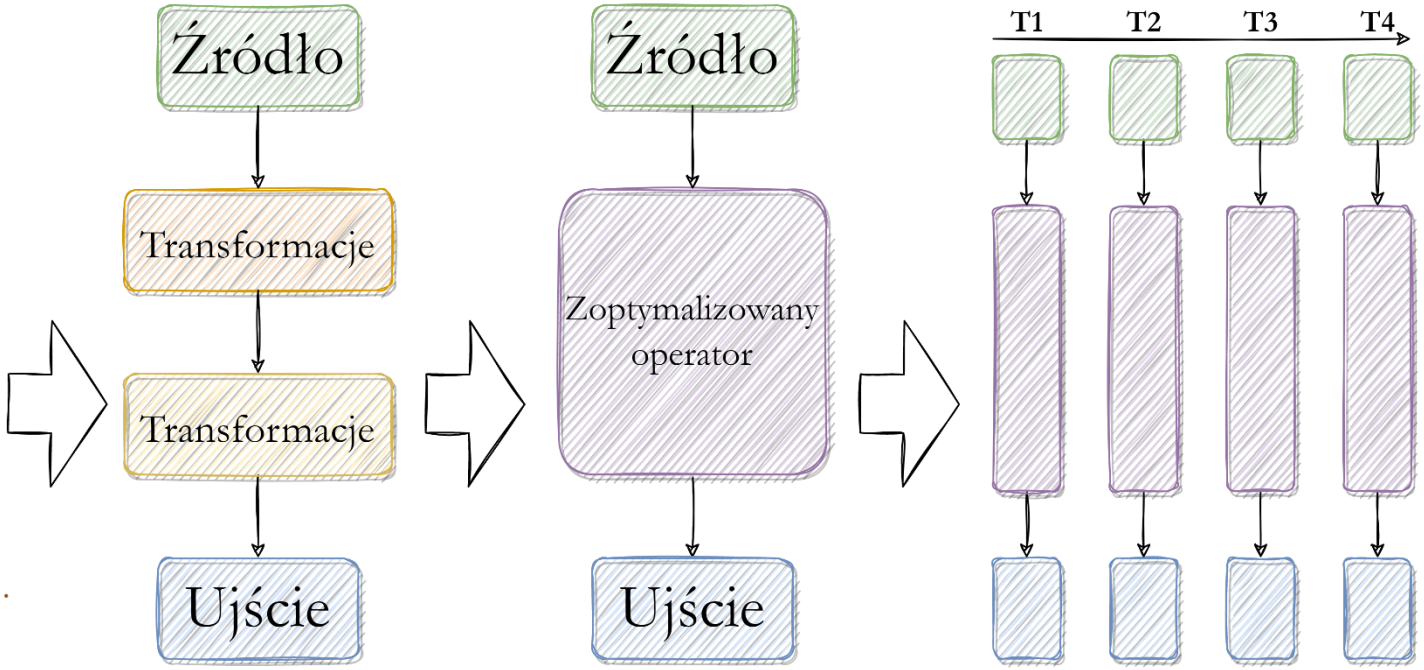
Transformacje
Przekształcenie danych wejściowych do docelowej (wynikowej) postaci

Ujście
Zapis przetransformowanego zapytania do systemu zewnętrznego

Szczegóły dotyczące przetwarzania
Definicja wyzwalacza
Tryb generowania danych wyjściowych
Lokalizacja punktu kontrolnego

Podsumowanie

```
val rates = spark.
  readStream.
  format("rate").
  option("rowsPerSecond", 1000000).
  load().
  withColumn("modulo", $value % 10).
  groupBy($"modulo").
  agg(sum($"value").as("sumval")).
  writeStream.
  format("memory").
  queryName("modulo_query").
  trigger("1 minute").
  outputMode("update").
  option("checkpointLocation",
         "/tmp/_checkpoints/modulo").
  start()
```



DataFrame, Dataset SQL

Plan logiczny

Plan zoptymalizowany

Seria inkrementacyjnych planów zapytania