Spark Structured Streaming
elementy zaawansowane

Krzysztof Jankiewicz

Plan

- Watermark – gwarancje
- Eliminowanie duplikatów
- Łączenie strumieni
- Operacje stanowe użytkownika
- Wskazówki dotyczące optymalizacji przetwarzania
- Obsługa awarii
- Monitorowanie stanu
- Diagnostyka
- RocksDB state store

Watermark – gwarancje

- Gwarancje dotyczące znaczników *watermark* są jednokierunkowe
  - Spark **gwarantuje**, że **nie usunie** stanu, który może zostać zmieniony w wyniku pojawiania się zdarzeń opóźnionych o mniej niż wynika to ze znacznika *watermark*
  - Jednocześnie, Spark **nie gwarantuje**, że **nie pozostawi** stanu, który mógłby być już usunięty. Innymi słowy zdarzenia spóźnione mogą (choć nie muszą) zostać uwzględnione podczas przetwarzania

Eliminowanie duplikatów a *watermarki*

- Metoda `dropDuplicates`
  - przepuszcza tylko pierwsze zdarzenie charakteryzujące się daną wartością wskazanego parametru

```
trips.dropDuplicates("tripID")
```

  - ma za zadanie nie dopuścić do ponownej obsługi tych samych zdarzeń (o ile znajdą się ponownie w strumieniu wejściowym)
  - przechowuje stan dla każdej wartości wskazanego parametru!
- Typowa logika biznesowa
  - duplikat zdarzenia może pojawić się w bardzo bliskiej odległości od zdarzenia pierwotnego,
  - nie ma niebezpieczeństwa, że to samo zdarzenie pojawi się po odpowiednio długim czasie – nie ma sensu przechowywanie stanu "starych" zdarzeń

```
trips.withWatermark("eventTime", "5 minutes").
          dropDuplicates("tripID", "eventTime")
```

Łączenie strumieni i *watermarki*

- Łączenie strumieni i statycznych zbiorów danych jest operacją bezstanową
- Analogiczne podejście do łączenia dwóch strumieni powodowałoby, że wynik byłby pusty (zdarzenia nie pojawiają się w tym samym momencie w obu strumieniach, tym bardziej zdarzenia spełniające warunki połączenia)
- Aby rozwiązać ten problem, łączenie dwóch strumieni obsługiwane jest jako operacja stanowa, w której utrzymywany jest stan obu strumieni – ostatnie zdarzenia zgodnie z *watermarkiem*

```
val transakcje = spark.readStream. ...
val kursyWalut = spark.readStream. ...

val transakcjeWithWatermark =
        transakcje.
            withWatermark("dataTransakcji", "2 hours")
val kursyWalutWithWatermark =
        kursyWalut.
            withWatermark("dataKursu", "2 days")

transakcjeWithWatermark.join(
 kursyWalutWithWatermark,
  expr("""
    kursWalutaId = transakcjaWalutaId AND
    dataTransakcji >= dataKursu AND
    dataTransakcji <= dataKursu + interval 1 day""")
)
```

definiujemy *watermarki* dla każdego strumieni aby utworzyć utrzymywany stan

definiujemy warunek połączeniowy oparty na

- znacznikach czasowych – 2 opcje
  - wykorzystanych do określenia watermarków
  - wynikających z wcześniej zdefiniowanych okien (opartych na znacznikach czasowych użytych w wm)
- logice biznesowej (opcjonalnie)

Warunek połączeniowy może jednoznacznie określać zdarzenia, które mogą opuścić stan – nie będzie on rósł w nieskończoność

Dwa typy operacji stanowych

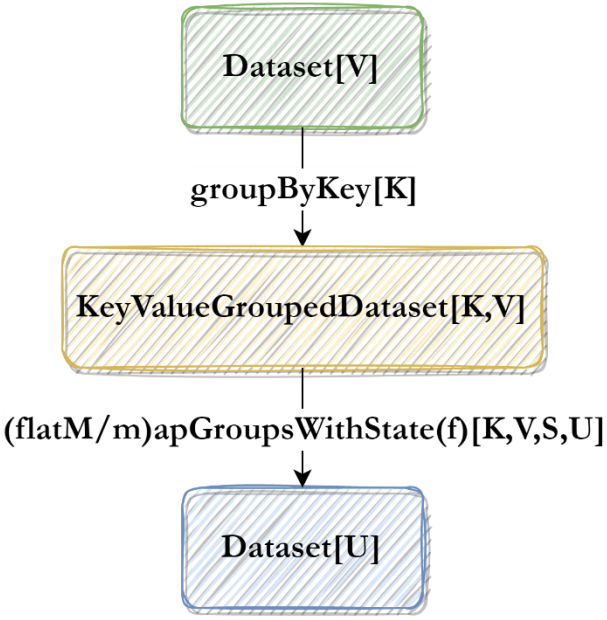
- Wbudowane
  - agregacje, eliminowanie duplikatów, połączenia
  - semantyka (SQL) jest dobrze zdefiniowania
  - utrzymywany stan czyszczony jest automatycznie
- Użytkownika
  - `mapGroupsWithState` oraz `flatMapGroupsWithState`
  - semantyka użytkownika
  - za czyszczenie stanu odpowiada użytkownik

- Wiele operacji na strumieniu danych wymaga logiki, która wykracza poza standardowe operatory.
  - Przykład 1:
    - dane wejściowe: kod lotniska, typ zjawiska pogodowego, siła zjawiska, czas, początek/koniec
    - oczekiwany wynik – dla każdego lotniska, chcemy znać jego aktualny status zjawisk pogodowych podzielony na dwie listy: zjawiska lekkie i poważne
  - Przykład 2:
    - dane wejściowe: identyfikator waluty, kurs waluty, czas
    - oczekiwany wynik: dla każdej waluty chcemy wiedzieć czy jej wartość spada czy rośnie

Operacje stanowe użytkownika

- Implementowane przy użyciu funkcji
  - `flatMapGroupsWithState`
  - `mapGroupsWithState`
- Operują na typie `KeyValueGroupedDataset`
  - `Dataset` dla
    - danych wejściowych
    - danych wynikowych
  - Konieczność definiowania typów danych
    - danych wejściowych – V
    - danych wynikowych – U
  - Działają na grupach wyznaczonych przez klucze
- Otrzymują jako argument funkcję mapującą – `f`
  - przekształcającą paczkę zdarzeń wejściowych
  - na zbiór (`flatMap`) lub pojedyncze (`map`) zdarzenie wynikowe
  - przy wykorzystywaniu utrzymywanego dla każdej grupy stanu – S
- Funkcje – różnica
  - `flatMapGroupsWithState` – zwraca iterator zdarzeń wynikowych
  - `mapGroupsWithState` – zwraca pojedyncze zdarzenie wynikowych

- Procedura
  1. Przygotowujemy postać strumienia
  2. Definiujemy struktury danych
     - Zdarzenia wejściowe – V
     - Postać stanu – S
     - Zdarzenia wynikowe – U
  3. Definiujemy funkcję mapującą
  4. Wykorzystujemy funkcję mapującą w metodzie `(flatM/m)apGroupsWithState`



Operacje stanowe użytkownika

przykład – struktury danych

```
val exchangeRatesDF = …
exchangeRates: org.apache.spark.sql.DataFrame

exchangeRatesDF.printSchema()
root
 |-- timestamp: timestamp (nullable = true)
 |-- currency: string (nullable = true)
 |-- rate: double (nullable = true)
```

- Definiujemy struktury danych
  - Zdarzenia wejściowe – V
  - Postać stanu – S
  - Zdarzenia wynikowe – U
- Konwersja `DataFrame` na `DataSet`
- Grupowanie danych w oparciu o klucz

```
case class CurrRate (
    currency: String,
    time: Long,
    rate: Double )
```

```
case class CurrStatus (
    currency: String,
    status: String
)
```

```
case class CurrLastRate (
    currency: String,
    var lastRate: Double,
    var status: String )
```

```
val exchangeRatesDS = exchangeRates.select($"currency", $"timestamp".as("time"), $"rate").as[CurrRate]

exchangeRatesDS: org.apache.spark.sql.Dataset[CurrRate]
```

```
val exchangeRatesDSGrouped = exchangeRatesDS.groupByKey(_.currency)

exchangeRatesDSGrouped: org.apache.spark.sql.KeyValueGroupedDataset[String,CurrRate]
```

Operacje stanowe użytkownika
metody mapujące

- Celem funkcji mapujących jest
  - obsługa zdarzeń wejściowych
  - z wykorzystaniem utrzymywanego stan
  - celem generacji zdarzeń wynikowych
- Parametry:
  - Identyfikator grupy
  - Zbiór zdarzeń wejściowych
  - Poprzedni stan dla grupy
- Wynik
  - dla `flatMapGroupsWithState` – iterator zdarzeń wynikowych
  - dla `mapGroupsWithState` – pojedyncze zdarzenie wynikowe
- Ciało
  - obsługujemy opcjonalny timeout stanu grupy
  - pobieramy poprzedni stan
  - wyliczamy nowy stan
  - aktualizujemy stan
  - opcjonalnie ustawiamy timeout grupy
  - wysyłamy zdarzenie lub zdarzenia wynikowe

Operacje stanowe użytkownika

przykład – metody mapujące, czas ważności

- dla `mapGroupsWithState`
- dla `flatMapGroupsWithState`

```
def map2Status(currency: String,
               currRates: Iterator[CurrRate],
               state: GroupState[CurrLastRate]): CurrStatus = {
    val lastS

    currRates
         if (
            l
         } el
            l
         }
         last
    }
    state.upd

    CurrStatu
}
```

```
def map2Statuses(currency: String,
                 currValues: Iterator[CurrRate],
                 state: GroupState[CurrLastRate]): Iterator[CurrStatus] = {
    var statuses = new ListBuffer[CurrStatus]()



        val lastState = state.getOption.getOrElse(CurrLastRate(currency,0,"~"))

        currValues.foreach{
            cv => if (cv.rate > lastState.lastRate) {
                lastState.status = "UP"
            } else {
                lastState.status = "DOWN"
            }
            lastState.lastRate = cv.rate
            statuses += CurrStatus(currency,lastState.status)
        }
        state.update(lastState)


    statuses.iterator
}
```

Operacje stanowe użytkownika
przykład – operatory stanowe

- `mapGroupsWithState`
- `flatMapGroupsWithState`

```
import org.apache.spark.sql.streaming.GroupStateTimeout

val result = exchangeRatesDSGrouped.mapGroupsWithState(
    timeoutConf = GroupStateTimeout.NoTimeout)(
    func = map2Status)
```

```
import org.apache.spark.sql.streaming.GroupStateTimeout
import org.apache.spark.sql.streaming.OutputMode

val result = exchangeRatesDSGrouped.flatMapGroupsWithState(
    outputMode = OutputMode.Append,
    timeoutConf = GroupStateTimeout.ProcessingTimeTimeout)(
    func = map2Statuses)
```

Parametry:

- `func` – funkcja mapująca
- `outputMode` – tryb generacji wyniku dla operatora (dostępne wartości to `Append` i `Update`)
  - *update mode* – zdarzenia wynikowe są parami klucz-wartość, każde zdarzenie wynikowe aktualizuje w tabeli wynikowej wartość o tym samym kluczu
  - *append mode* – zdarzenia wynikowe są niezależne i dodawane są do tabeli wynikowej
- `timeoutConf` – konfiguracja wskazująca typ znaczników czasu na podstawie którego określany jest czas ważności dla stanu w grupie. Dostępne wartości:
  - `GroupStateTimeout.ProcessingTimeTimeout`
  - `GroupStateTimeout.NoTimeout`
  - `GroupStateTimeout.EventTimeTimeout`

Czas ważności – dlaczego?

- Jeśli:
  - wszystkie zjawiska pogodowe się zakończyły – obie listy są puste
  - przez dobę nie dostajemy żadnych dodatkowych informacji,
- to chcemy usunąć informację o tym lotnisku z naszego stanu (optymalizacja przetwarzania)

Odporność na awarie – aplikacje 24/7

- Aplikacje oparte na strumieniach danych działają w trybie 24/7, w związku z czym muszą być one wyjątkowo odporne na błędy (systemowe, JVM itp.)
- Aby to uzyskać aplikacje **zapisują** pewne informacje na **nośnikach** zdolnych przetrwać awarie. Taki zapis nosi nazwę **punktu kontrolnego**.
- Punkty kontrolne wykorzystywane są w szczególności z dwóch powodów:
  - aby **materializować stan obliczeń** w celu uniknięcia czasochłonnego przeliczania tego stanu z danych źródłowych
  - umożliwić ponowne uruchomienie programu sterownika, który na podstawie zapisów w punkcie kontrolnym będzie wiedział jak odtworzyć stan przetwarzania i go kontynuować.
- W związku z powyższym rozróżnia się dwa typy punktów kontrolnych:
  - punkty kontrolne **metadanych** (*metadata checkpointing*), zawierające:
    - dane dotyczące konfiguracji – wymagane do restartu aplikacji
    - operacje DStream – zbiór operacji definiujących aplikację przetwarzającą DStreams
    - "niedokończone" porcje danych – porcje danych, których przetwarzanie nie zostało zakończone
  - punkty kontrolne **danych** (*data checkpointing*), zawierające pośrednie etapy przetwarzania transformacji stanowych, dzięki czemu znacząco może zostać ograniczona:
    - liczba źródłowych porcji danych, oraz
    - czas i zasoby

    Punkty kontrolne danych wymagane są do odzyskania stanu przetwarzania

Obsługa awarii

```
aggDF
  .writeStream
  .outputMode("complete")
  .option("checkpointLocation", "path/to/HDFS/dir")
  .format("memory")
  .start()
```

Optymalizacja przetwarzania

- Liczba partycji 1-3 na rdzeń
  - zbyt mało – mała przepustowość – wolne rdzenie
  - zbyt dużo – konieczność zapisu stanu przetwarzania na HDFS – większa latencja
- Uwaga na wielkość utrzymywanych stanów na jednego wykonawcę
  - im większy stan, tym dłuższy czas wykonywania punktów kontrolnych, JVM GC

Monitorowanie stanu – programowo

- synchroniczne

```
val query: StreamingQuery = ...

println(query.lastProgress)
println(query.status)
```

- asynchronicznie

```
val spark: SparkSession = ...

spark.streams.addListener(new StreamingQueryListener() {
    override def onQueryStarted(queryStarted: QueryStartedEvent): Unit = {
        println("Query started: " + queryStarted.id)
    }
    override def onQueryTerminated(queryTerminated: QueryTerminatedEvent): Unit = {
        println("Query terminated: " + queryTerminated.id)
    }
    override def onQueryProgress(queryProgress: QueryProgressEvent): Unit = {
        println("Query made progress: " + queryProgress.progress)
    }
})
```

Patrz: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Monitorowanie stanu – programowo

```
println(query.lastProgress)
{
  "stateOperators" : [ {
    "numRowsTotal" : 10,
    "numRowsUpdated" : 10,
    "memoryUsedBytes" : 82800,
    "numRowsDroppedByWatermark" : 0,
    "customMetrics" : {
      "loadedMapCacheHitCount" : 70000,
      "loadedMapCacheMissCount" : 0,
      "stateOnCurrentVersionSizeBytes" : 19360
    }
  } ],
  "sources" : [ {
    "description" : "RateStreamV2[rowsPerSecond=1000000,
                                  rampUpTimeSeconds=0, numPartitions=default",
    "startOffset" : 369,
    "endOffset" : 371,
    "numInputRows" : 2000000,
    "inputRowsPerSecond" : 880669.3086745928,
    "processedRowsPerSecond" : 895255.1477170994
  } ],
  "sink" : {
    "description" : "MemorySink",
    "numOutputRows" : 10
  }
}
```

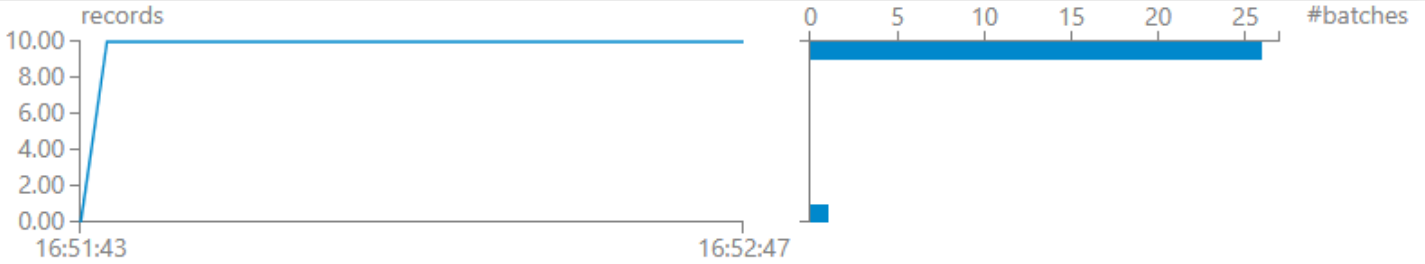
Inne metody `StreamingQuery` i `SparkSession`

- `StreamingQuery`:
  - `query.id` – unikalny identyfikator uruchomionego zapytania, który jest stały po ponownym uruchomieniu z danych punktu kontrolnego
  - `query.runId` – unikalny identyfikator tego uruchomienia zapytania, który będzie generowany przy każdym uruchomieniu / restarcie
  - `query.name` – nazwa zapytania nadana przez użytkownika lub automatycznie wygenerowana
  - `query.explain()` – dostarcza plan zapytania
  - `query.stop()` – zatrzymuje zapytanie
  - `query.awaitTermination()` – blokuje dalsze przetwarzanie aż do momentu zatrzymania zapytania za pomocą metody stop() lub zgłoszenia wyjątku
  - `query.exception` – wyjątek jeśli zapytanie zakończyło się błędem
  - `query.recentProgress` – tablica ostatnich postępów zapytania
  - `query.lastProgress` – informacje na temat ostatniego postępu zapytania
- `SparkSession`
  - `spark.streams.active` – pobiera listę aktywnych zapytań strumieniowych
  - `spark.streams.get(id)` – pobiera obiekt zapytania na podstawie jego id
  - `spark.streams.awaitAnyTermination()` – blokuje przetwarzania do końca dowolnego z zapytań

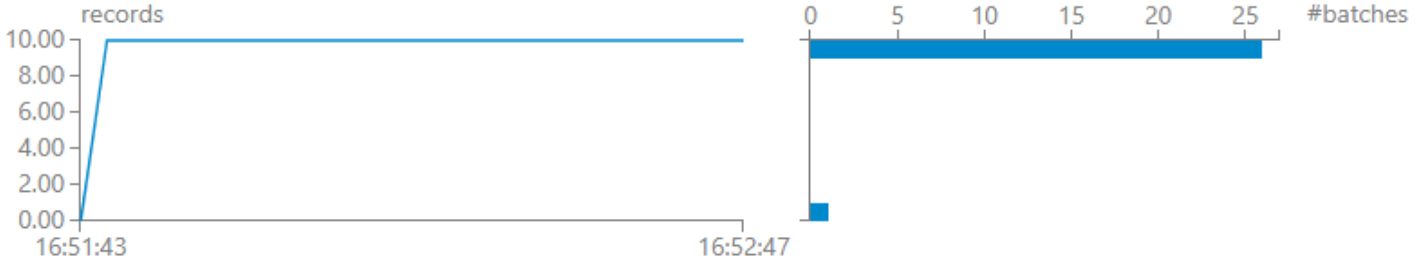
```
spark.streams.active.foreach(_.stop)
```

Monitorowanie stanu – interfejs sieciowy

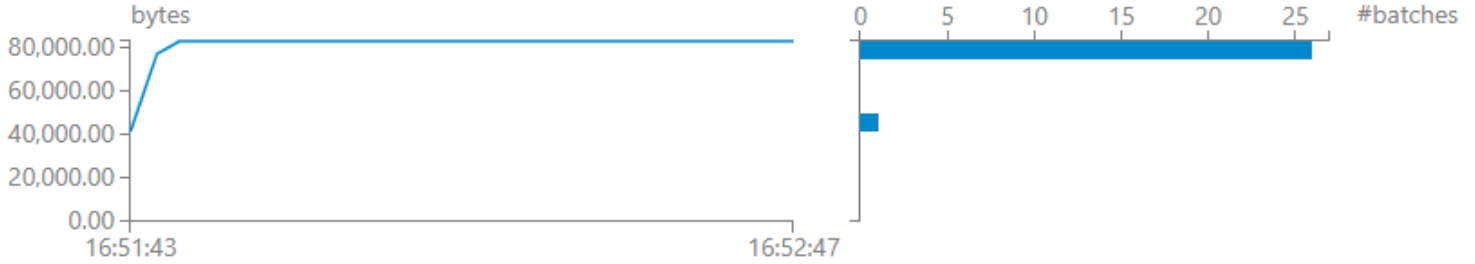
Aggregated Number Of Total State Rows (?)



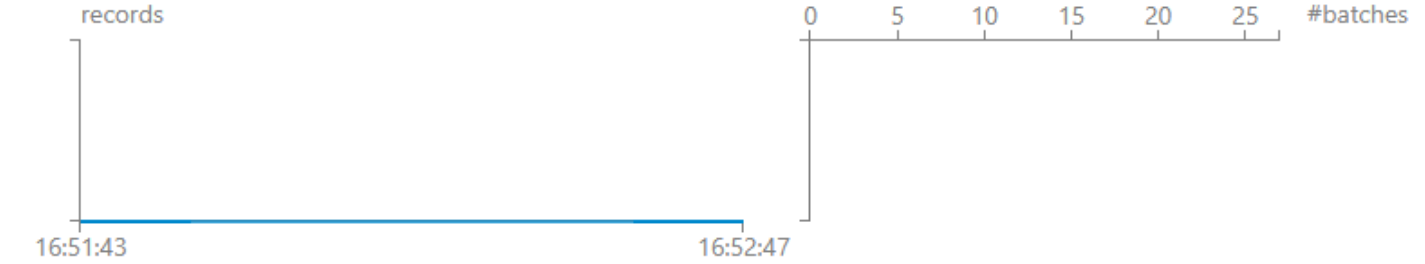
Aggregated Number Of Updated State Rows (?)



Aggregated State Memory Used In Bytes (?)



Aggregated Number Of Rows Dropped By Watermark (?)

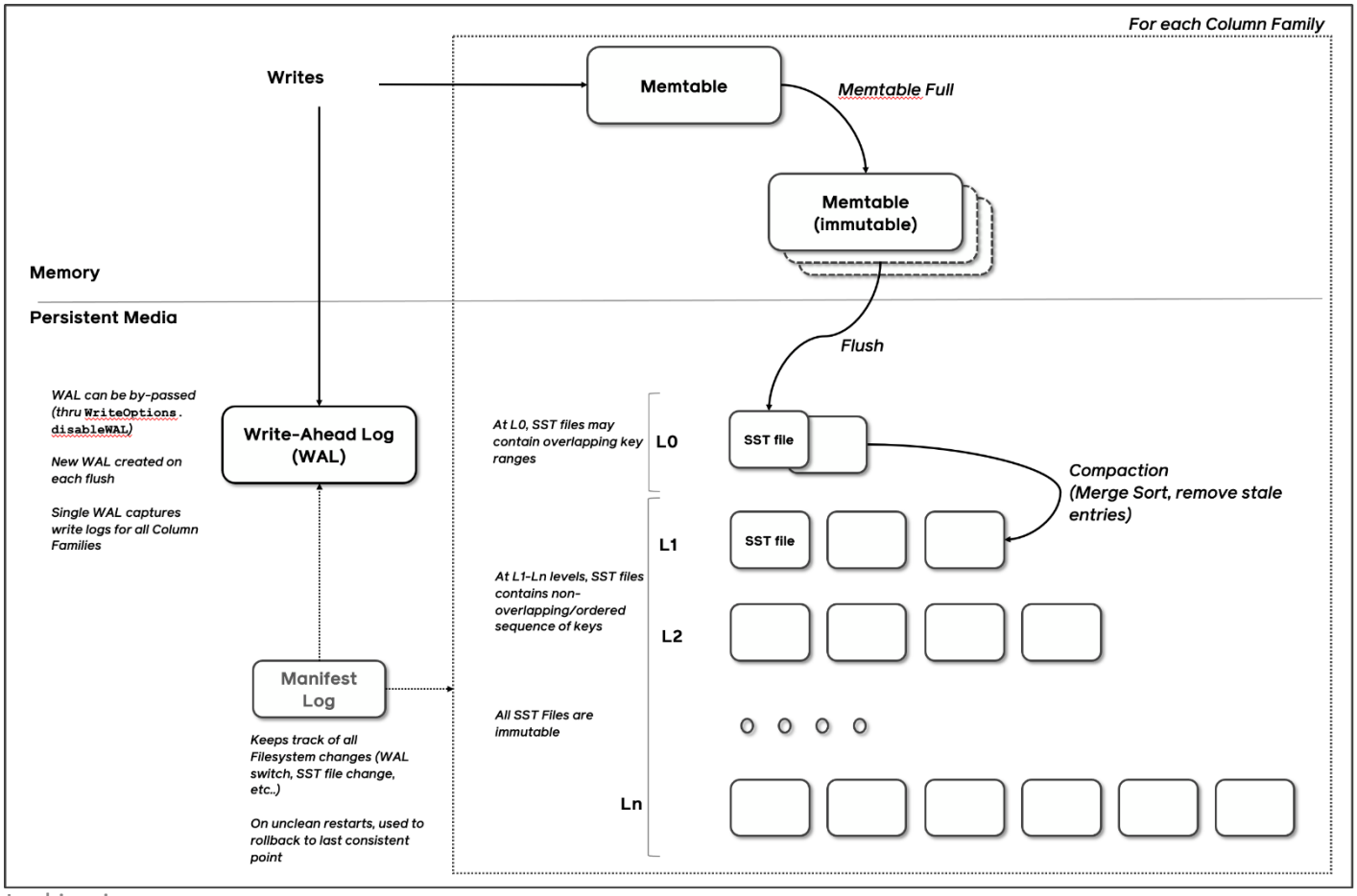


Diagnostyka

- Interfejs sieciowy Sparka pozwala na
  - weryfikację wielkości stanów obsługiwanych przez wykonawców (brak zrównoważenia)
  - wykrywanie wolnych operacji usuwania krotek
  - wykrywanie długotrwałych operacji punktu kontrolnego

RocksDB state store

- Funkcjonalność dostępna od wersji 3.2.0
- Czym jest RockDB
  - wbudowana baza danych klucz-wartość
  - napisana w C++ celem maksymalnej wydajności
- Zalety
  - Brak konieczności wykorzystywania sterty JVM
  - Brak GC
  - Stan jest utrwalany automatycznie w HDFS
  - Gwarancje exactly-once



Patrz: https://github.com/facebook/rocksdb/wiki/RocksDB-Overview#3-high-level-architecture

Pytania

- Czy agregacja wymaga utrzymywania stanu w tabeli wynikowej?
- Jakie inne poznane operatory wymagają utrzymywania stanu?
- Załóżmy, że dokonujemy grupowania i agregacji
  - Jaki wpływ ma definicja okna na liczbę (zwiększanie/zmniejszanie) wierszy tabeli wynikowej (na utrzymywany stan)?
    Rozważ przypadki:
    - `groupBy(window($"ts", "10 minutes"), $"clientIpAddress").agg(sum($"bytesSent"), count($"bytesSent"))`
    - `groupBy($"clientIpAddress").agg(sum($"bytesSent"), count($"bytesSent"))`
  - Czy taki wpływ ma tylko definicja okna?
    Czy może także grupowanie po wybranych atrybutach?
  - Czy w każdym trybie generowania wyniku jest tak samo?

Podsumowanie

- Watermark – gwarancje
- Eliminowanie duplikatów
- Łączenie strumieni
- Operacje stanowe użytkownika
- Wskazówki dotyczące optymalizacji przetwarzania
- Monitorowanie stanu
- Diagnostyka
- RocksDB state store

- Dla tych, którzy czują niedosyt: `https://jaceklaskowski.github.io/spark-structured-streaming-book/`

Po podsumowaniu

- Czy *Spark Structured Streaming* to ideał?
- Idea *tworzę aplikację wsadowo => dostaję strumieniową*
  wydaje się genialna
- A jednak
  - to jej konsekwencją jest tak wiele różnorodnych ograniczeń, np. brak implementacji łańcuchów agregacji
  - jest niespójna
    - przetwarzanie stanowe użytkownika
      - znaczniki *watermark*
      - tryb generacji wyniku do dalszego przetwarzania (`update`, append)
    - łączenie strumieni danych
      - znaczniki *watermark*
      - wymuszenie trybu append
- Czy jest rozwiązanie?
  Wydaje się, że ta idea będzie musiała być (lub już jest) zmodyfikowana.
- Jak?
  Każdy operator stanowy (jak te powyżej) powinien określać sposób obsługi wyniku (konsekwencja zmian stanu)
- Konsekwencje:
  - kończymy z idealną ideą – coś co odróżnia StrStr od innych rozwiązań
  - znacząco komplikujemy przetwarzanie