

BIG DATA APACHE SPARK NOTATKI NA START

- Big Data
- Architektura Spark
- Partycjonowanie
- Optymalizacja
- Dataframe
- Przesyłanie Strumieniowe
- Monitorowanie
- Ciekawe blogi

CEGLADANYCH.PL

Autor

Krzysztof Nojman

| | |
|--|-----------|
| Jak kwalifikować dane do Big Data 3V | 1 |
| Architektura Apache Spark | 2 |
| Komponenty Apache Spark | 2 |
| <u>Sterownik (sparkcontext)</u> | <u>2</u> |
| <u>Kierownik klastra (cluster manager)</u> | <u>2</u> |
| <u>Wykonawcy (Executors)</u> | <u>2</u> |
| <u>Pamięć wykonawcy</u> | <u>3</u> |
| Aplikacja Spark | 4 |
| <u>Spark API</u> | <u>5</u> |
| <u>Interfejsy API niższego poziomu</u> | <u>6</u> |
| <u>Strukturalny interfejs API</u> | <u>6</u> |
| <u>Rozproszona architektura</u> | <u>6</u> |
| <u>Jobs Stages Tasks</u> | <u>7</u> |
| <u>Tolerancja błędów</u> | <u>7</u> |
| Partycjonowanie | 7 |
| <u>Partycjonowanie DataFrame prosto ze źródła danych</u> | <u>8</u> |
| <u>Zmiana partycji coalesce() vs repartition()</u> | <u>11</u> |
| <u>Kontrolowanie liczby partycji</u> | <u>11</u> |
| Optymalizacja | 12 |
| <u>Co można optymalizować</u> | <u>12</u> |
| <u>Równoległość</u> | <u>14</u> |
| <u>Skalowanie aplikacji</u> | <u>14</u> |
| <u>Konfiguracja Spark</u> | <u>14</u> |
| <u>Optymalizator Catalyst</u> | <u>15</u> |
| <u>Plan Logiczny i Fizyczny</u> | <u>15</u> |
| <u>Identyfikowanie wąskich gardel wydajności w aplikacjach Spark</u> | <u>16</u> |
| <u>Plan wykonania</u> | <u>16</u> |
| <u>cache() alias persist()</u> | <u>16</u> |
| <u>Kiedy użyć Cache i persist</u> | <u>17</u> |
| <u>Kiedy nie używać</u> | <u>17</u> |
| <u>Poziomy magazynowania danych</u> | <u>17</u> |
| <u>Adaptacyjne wykonywanie zapytań</u> | <u>18</u> |
| Spark Dataframe | 18 |
| <u>Podstawowe typy danych</u> | <u>18</u> |
| <u>Czas uniksowy</u> | <u>20</u> |
| <u>Skomplikowane Typy Danych</u> | <u>21</u> |
| <u>ArrayType-MapType-StructType</u> | <u>22</u> |
| <u>Schematy Danych</u> | <u>23</u> |
| <u>Kiedy używamy schematu</u> | <u>23</u> |
| <u>DataFrame i zestawy danych</u> | <u>24</u> |
| <u>Bezpieczne ładowanie danych</u> | <u>24</u> |

| | |
|--|-----------|
| <u>Tryby Zapisu (Save Modes)</u> | 26 |
| <u>DataSet</u> | 27 |
| <u>Encoders</u> | 27 |
| <u>Tworzenie Dataset</u> | 28 |
| <u>Transformacje</u> | 28 |
| <u>Łączniki</u> | 29 |
| <u>Grupowanie i agregację</u> | 29 |
| <u>Kiedy użyć DataFrame a kiedy Dataset</u> | 29 |
| <u>Spark SQL</u> | 30 |
| <u>Tabele</u> | 31 |
| <u>Widoki</u> | 32 |
| <u>Apache Hive</u> | 33 |
| <u>Katalog</u> | 35 |
| <u>Odczyt i zapis DataFrames</u> | 35 |
| <u>Operacje na kolumnach</u> | 36 |
| <u>Zarządzanie nulls</u> | 37 |
| <u>Transformacje i akcje</u> | 38 |
| <u>Transformacje Wide – Narrow</u> | 39 |
| <u>Shuffle</u> | 40 |
| <u>Pipelining</u> | 40 |
| <u>Łączniki (Joins)</u> | 40 |
| <u>Hints (podpowiedzi)</u> | 42 |
| <u>Typy Łączników</u> | 42 |
| <u>Duplikaty kolumn</u> | 42 |
| <u>UDFs Funkcje zdefiniowane przez użytkownika</u> | 43 |
| <u>Funkcje</u> | 45 |
| <u>Funkcje okienkowe</u> | 45 |
| <u>Przesyłanie strumieniowe</u> | 47 |
| <u>Porównanie</u> | 47 |
| <u>Strumienie ustrukturyzowane</u> | 47 |
| <u>Problem ze Spark Streaming - Micro-batch</u> | 47 |
| <u>Przetwarzanie micro-batcha</u> | 48 |
| <u>Opóźnienie</u> | 48 |
| <u>Mikro-batch alternatywa</u> | 48 |
| <u>Zapytania do tabeli</u> | 49 |
| <u>Tryby wyjściowe (Output Modes)</u> | 50 |
| <u>Stream części składowe</u> | 50 |
| <u>Output Sinks</u> | 51 |
| <u>Charakterystyka Streamu</u> | 51 |
| <u>Odporność na błędy (Fault tolerance)</u> | 52 |
| <u>Schemat</u> | 52 |
| <u>Typy okienek Spark</u> | 55 |
| <u>Usuwanie duplikatów</u> | 57 |
| <u>Zabronione operacje</u> | 58 |

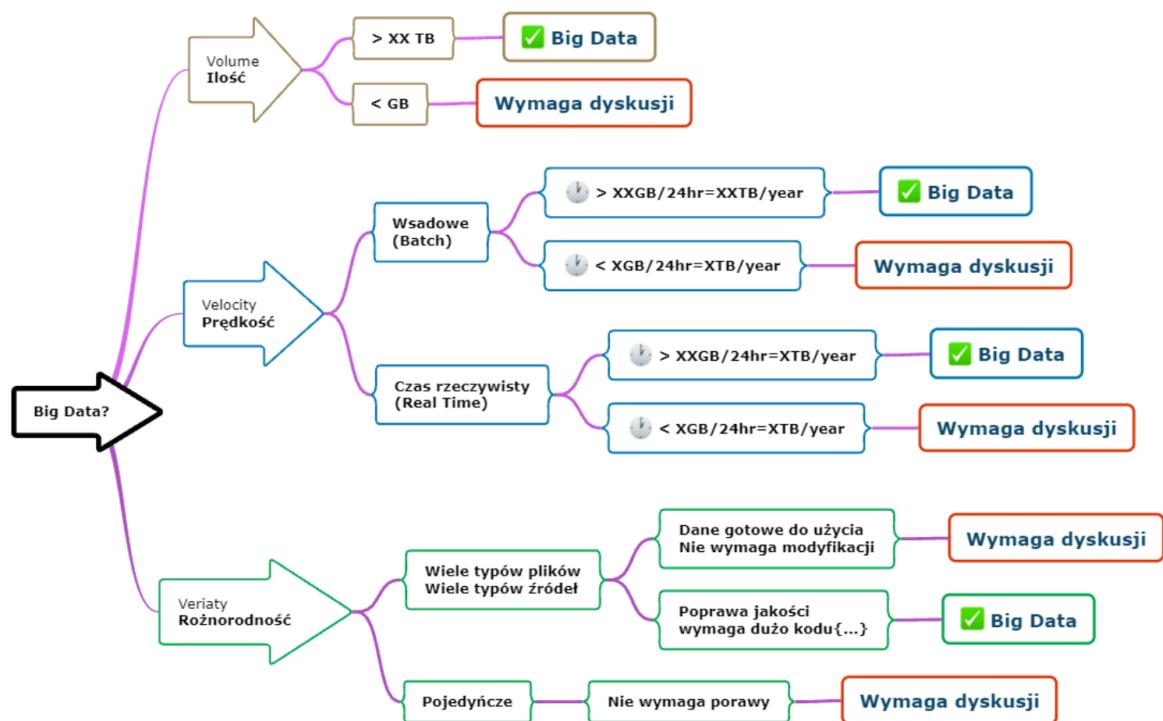
| | |
|---|-----------|
| <u>Odzyskiwanie po awarii - Checkpointing</u> | 58 |
| <u>Przykłady Spark Streaming</u> | 59 |
| Monitorowanie | 60 |
| <u>Monitoruj Driver</u> | 61 |
| <u>Logi Sparka</u> | 61 |
| <u>Spark UI</u> | 61 |
| <u>Statystyki Stage</u> | 61 |
| <u>Jobs</u> | 63 |
| <u>Wolne Taski</u> | 64 |
| <u>Wolne agregacje</u> | 64 |
| <u>Wolne joiny</u> | 64 |
| <u>Wolne Odczyty i Zapisy</u> | 64 |
| <u>Błędy OutOfMemory</u> | 65 |
| Książki | 65 |
| Ciekawe Blogi | 65 |
| Lakehouse | 66 |
| Dobre praktyki Databricks | 67 |

Jak kwalifikować dane do Big Data 3V

Główną ideą tej technologii jest możliwość równoległego przetwarzania danych. Oznacza to, że możesz podzielić dane na partycje, czyli na małe kawałki. Tak podzielone, pozwalają na wykonanie kalkulacji na każdej partycji osobno.

Oczywiście nie ty to robisz, ale algorytmy Apache Spark, o którym możesz poczytać [tutaj](#). Np. Jeśli twoje dane są podzielone na 100 partycji, to narzędzia Big Data mogą przetwarzać **dane z każdej partycji w tym samym czasie**. To tak jakbyś oglądał 10 seriali na 10 telewizorach w tym samym czasie, jedna godzina i już jesteś na bieżąco z Netflixem.

- Zbliżamy się do kluczowego elementu, który pomoże w decyzji czy potrzebujesz technologii Big Data. Są to słynne **3 V**
 - **Volume** (Ilość)
 - **Velocity** (Prędkość)
 - **Veriaty** (Różnorodność)
- **V** (Ilość) + **V** (Prędkość)+ **V** (Różnorodność) = 3

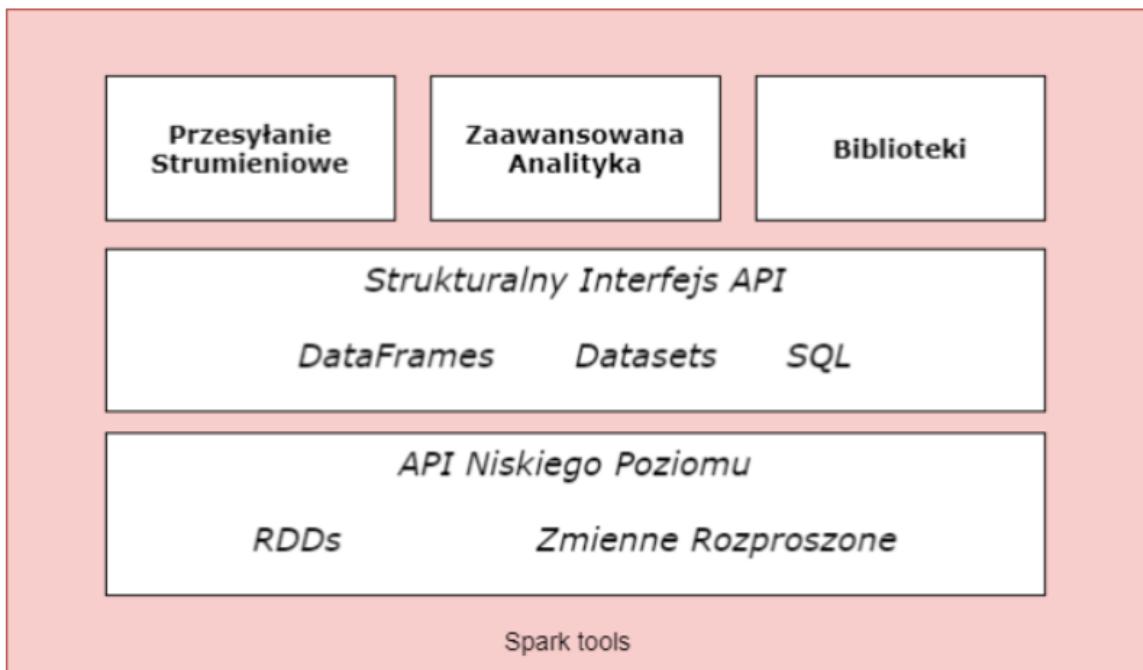


Link:

[Kiedy użyć Big Data - Ceglądanych \(cegladanych.pl\)](http://cegladanych.pl)

Architektura Apache Spark

Komponenty Apache Spark



Sterownik (sparkcontext)

Sparkcontext, jest to główny obiekt Sparda. Zajmuje się on koordynacją procesów aplikacji.

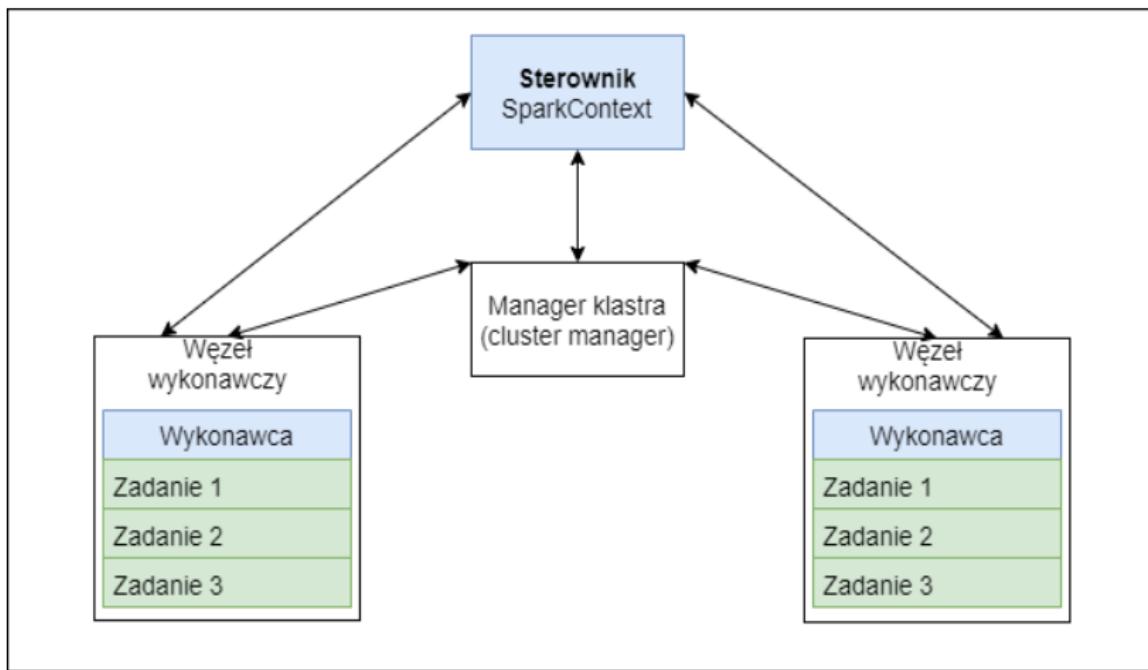
Kierownik klastra (cluster manager)

Nasz Sparkcontext łączy się z kilkoma managerami klastrów:
Ich zadaniem jest przypisanie zasobów dla aplikacji.

Wykonawcy (Executors)

Sterownik wysyła zadania do uruchomienia. Klauster manager komunikuje się z programami wykonawczymi i przydziela im zasoby.

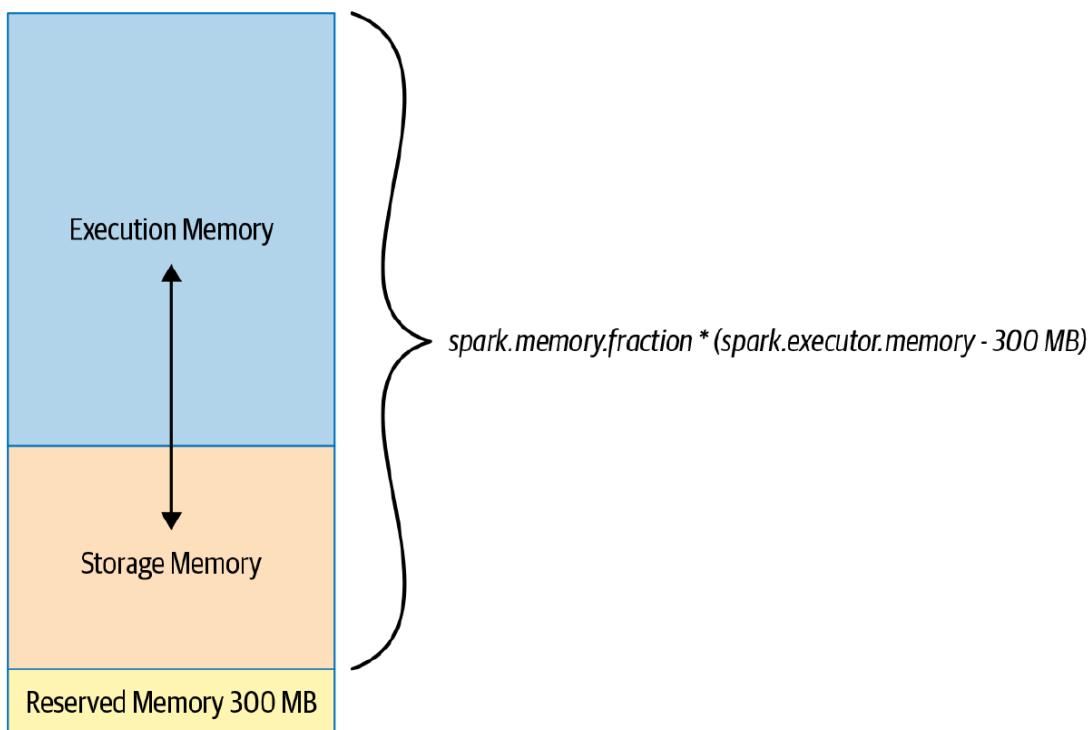
Każdy wykonawca jest odpowiedzialny za uruchomienie kodu i raportowanie stanu wykonania do sterownika.



Pamięć wykonawcy

Pamięć jest podzielona na część używaną przez wykonawcę 60% i magazyn danych 40%, oraz 300 MB jest zarezerwowane, żeby powstrzymać błędy **Out Of Memory errors**.

Ilość pamięci jest konfigurowalna - **spark.executor.memory**



Linki:

- <https://spark.apache.org/docs/latest/tuning.html#memory-management-overview>
- [MemoryManager - The Internals of Apache Spark \(japila.pl\)](#)

Aplikacja Spark

Izolacja

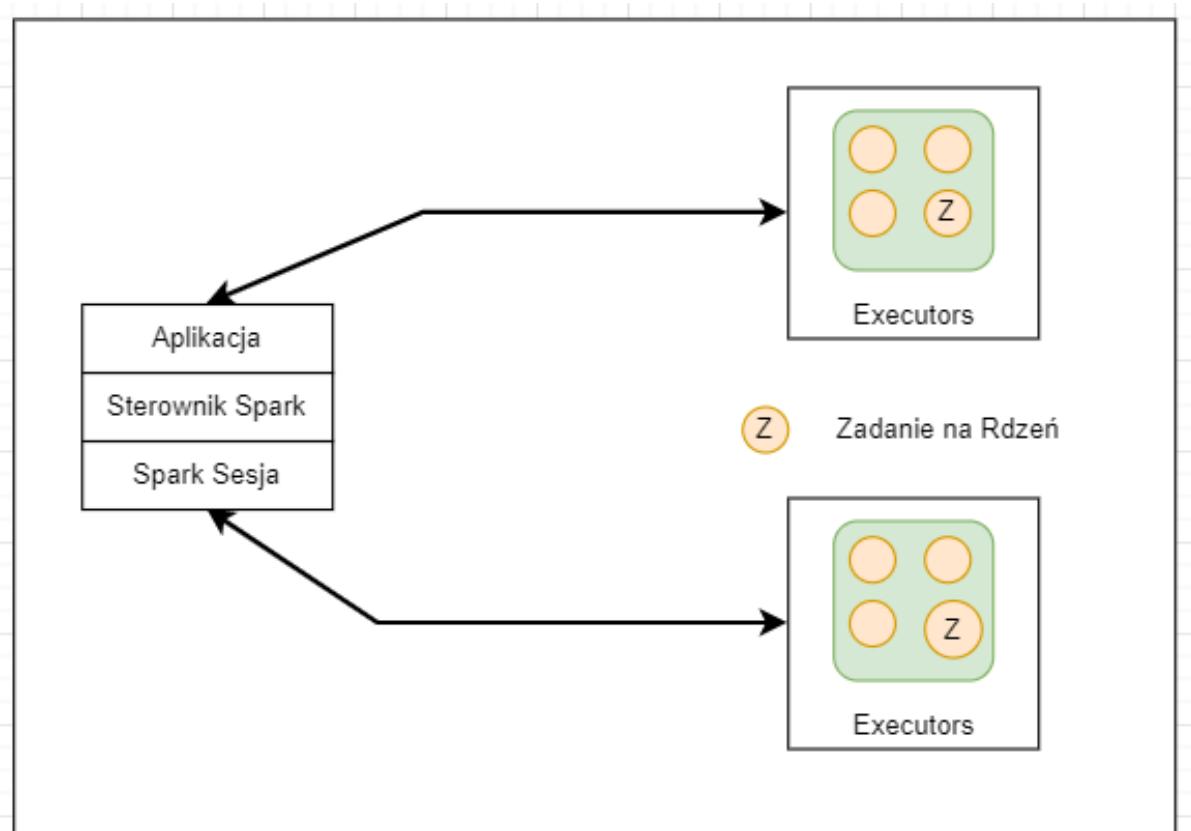
Każda aplikacja ma swój własny proces wykonawczy działający wielowątkowo. Trzeba dodać, że aplikacje działają w izolacji. Każdy sterownik planuje swoje własne zadania. Po stronie wykonawczej każda aplikacja działa na osobnej maszynie JVM. Zapewnia to izolację aplikacji i procesów a tym samym ma wpływ na stan systemu i jego osiągi.

Niezależność

Spark jest niezależny od głównego menedżera klastra. Może się on komunikować z wykonawcami.

Sieć

Sterownik musi być w ciągłej komunikacji z wykonawcami, a tym samym z siecią węzłów roboczych. Najkorzystniej jest, aby sterownik był w tej samej sieci lokalnej.



Linki

- <https://spark.apache.org/docs/latest/cluster-overview.html>
- <https://techvidvan.com/tutorials/spark-modes-of-deployment/>

Spark API

Strukturalne API nazywane są zestawami danych („dataset”). Możesz ich używać do pisania statycznego kodu w Javie lub Scali. Ten zestaw danych nie jest dostępny w języku Python lub R, ponieważ są to języki dynamiczne. Interfejs API zestawu danych umożliwia przypisywanie Java / Scala do rekordów w DataFrame i manipulowanie nimi jako kolekcją obiektów. Jeśli masz bardziej złożony problem, możesz go rozwiązać za pomocą interfejsów API niższego poziomu, takich jak zestawy danych („datasets”) lub RDD.

Przesyłanie strumieniowe

Największą zaletą jest szybkie wydobycie wartości ze strumienia praktycznie bez zmian kodu. Możesz rozpocząć od zadania wsadowego, a następnie przekonwertować go na streaming. W ten sposób poprzez stopniowe przetwarzanie danych masz pełną kontrolę nad całym procesem.

Nauczanie maszynowe

Masz dostęp do bardzo zaawansowanego interfejsu API, który pozwala na wykonanie kilku zadań, takich jak klasyfikacja, regresja, grupowanie lub głębokie uczenie się.

Interfejsy API niższego poziomu

Spark ma interfejsy API niższego poziomu, aby umożliwić dowolną manipulację obiektami Java i Python za pomocą Resilient Distributed Datasets (RDD). Prawie wszystko w Spark jest oparte na RDD. W przypadku podstawowych zadań zaleca się stosowanie DataFrames

Strukturalny interfejs API

Istnieją trzy rodzaje interfejsów API

- Zestawy danych
- DataFrames czyli ramka danych
- Tabele i widoki SQL

Dzięki strukturalnym interfejsom API można uruchamiać zarówno zadania wsadowe (batch), jak i strumieniowe (streaming). Zadania rozpoczynają się od wykonania wykresu instrukcji. Są one podzielone na etapy. Zestaw danych jest logiczną strukturą, którą można manipulować za pomocą przekształceń (transformacji) i działań (akcji).

Rozproszona architektura

Aplikacja Spark składa się z procesu sterownika oraz procesów wykonawczych. Proces sterownika uruchamia główną funkcję main()

Wykonawcy

Sterownik wysyła zadania do uruchomienia. Klauster manager komunikuje się z programami wykonawczymi i przydziela im zasoby. Spark również komunikuje się bezpośrednio z węzłami klastów (nodes). Każdy **wykonawca** jest odpowiedzialny za uruchomienie kodu i raportowanie stanu wykonania do sterownika.

Manager klastra kontroluje maszyny i przypisane im zasoby, żeby efektywnie wykonać aplikacje Sparka. Spark może być zarządzany przez: Spark's Standalone cluster manager, YARN, or Mesos.

Jobs (zadania)

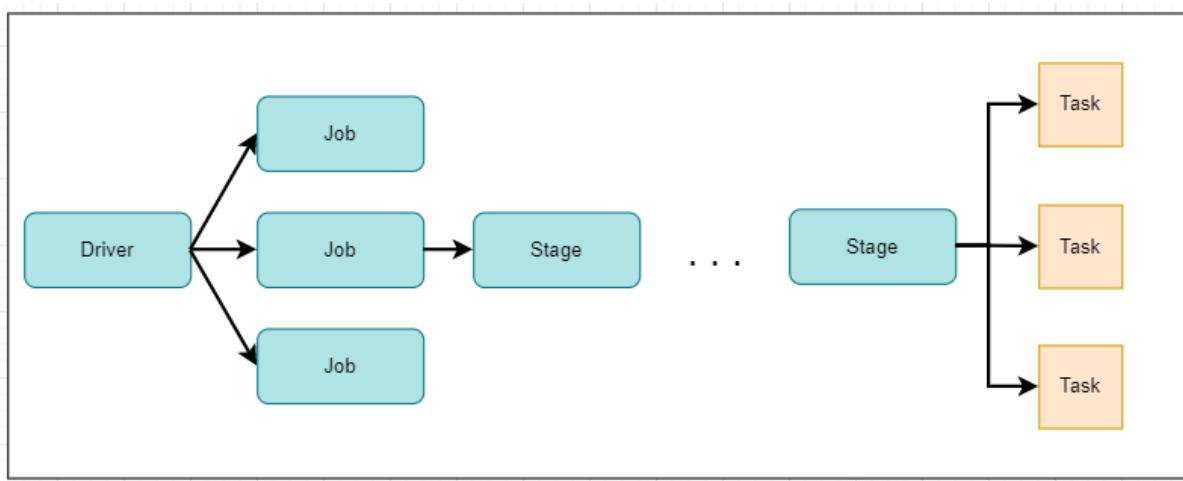
Spark dzieli wszystkie operacje na etapy (**JOBS>STAGE>TASK**). Pierwszy etap to **Job**, czyli zadanie są one uruchamiane przez akcje np. count(), saveAsTextFile(). W

tym etapie powstaje DAG, jest to plan wykonania. Od tego, jak wygląda DAG zależy jakie operacje Spark musi wykonać i na jakiej ilości partycji.

Następnie każdy Job będzie podzielony na **Stage** tutaj Spark podejmie decyzję co wykonać szeregowo a co równolegle. Jak się domyślasz nie wszystkie operacje da się wykonać od razu i trzeba je rozdzielić na logiczne etapy.

Ostatni etap to **Task**, jest on wykonywany na poszczególnych rdzeniach wykonawców. Odpowiada on jednej partycji wykonanej na pojedynczym rdzeniu.

Jobs Stages Tasks



Tolerancja błędów

Lineage + Immutability = Fault Tolerance

Spark zapisuje poszczególne etapy transformacji i jest w stanie odtworzyć stan poszczególnych etapów w chwili awarii. Jest to zasługa DAG, który dzieli wszystkie obliczenia na etapy.

Linki

- [Caching and Persistence - The Internals of Apache Spark \(japila.pl\)](#)
 - <https://docs.databricks.com/delta/optimizations/delta-cache.html>

Partycjonowanie

Podstawą technologii Big Data jest równoległe przetwarzanie danych oparte na partycjonowaniu. Oznacza to, że możesz podzielić dane na partie, czyli na małe kawałki.

Tak podzielone, pozwalają na wykonanie kalkulacji na każdej partycji osobno. Oczywiście nie ty to robisz, ale algorytmy Apache Spark, o których możesz poczytać

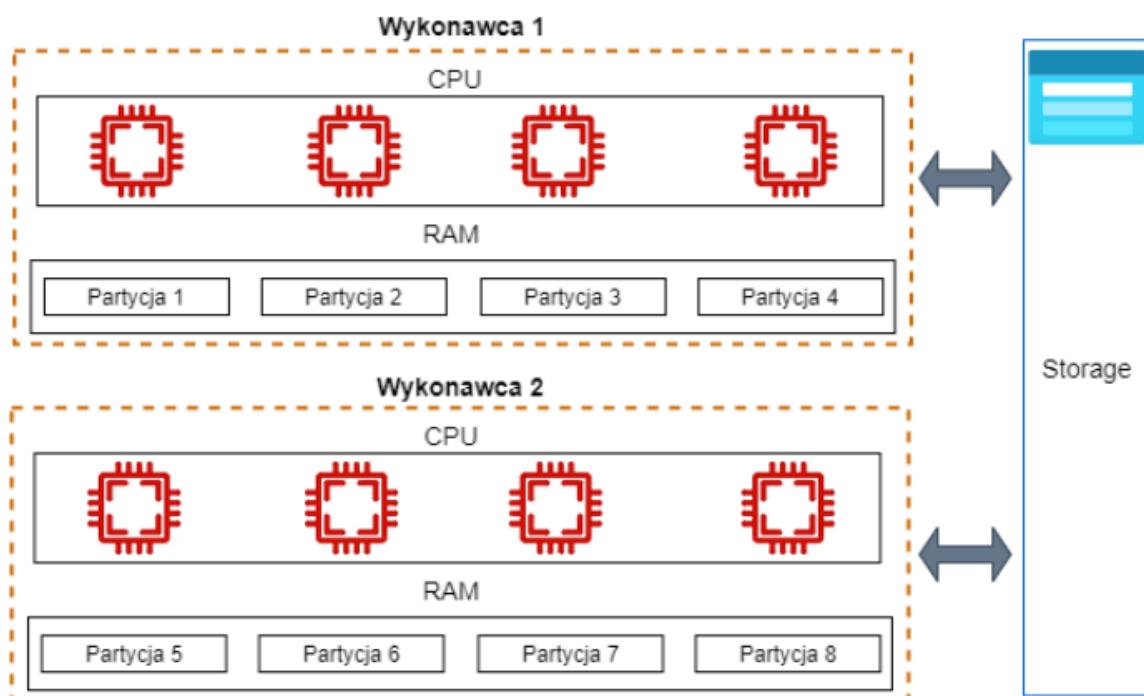
[tutaj](#). Np. Jeśli twoje dane są podzielone na 100 partycji, to narzędzia Big Data mogą przetwarzać dane z każdej partycji w tym samym czasie. To tak jakbyś oglądał 10 seriali na 10 telewizorach w tym samym czasie, jedna godzina i już jesteś na bieżąco z Netflixem.

Spark równoległość thread > task (core)

Task = 1 partycja.

Optymalizacja

Maksymalizacja równoległości: Spark może przetwarzać tyle partycji ile jest rdzeni (slotów) na wykonawcach.



Partycjonowanie DataFrame prosto ze źródła danych

Źródła danych

- CSV
- JSON
- Parquet
- ORC
- Połączenia JDBC / ODBC
- Zwykłe pliki tekstowe
- Społeczność zapewnia znacznie więcej połączeń, takich jak Cassandra, HBase, AWS Redshift, XML ect.

Odczyt interfejsu API wygląda następująco

`DataFrameReader.format(...).option("key", "value").schema(...).load()`

Podstawowa struktura zapisu danych jest następująca:

`DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...).sortBy(...).save()`

Jak powstaje partycja

Dane na dysku są ułożone w bloki.

Bloki są uzależnione od technologii – wartości często spotykane to 64MB do 256 MB

HDFS i AWS S3 to 128 MB

Azure:

| Service version | Maximum block size (via Put Block) | Maximum blob size (via Put Block List) | Maximum blob size via single write operation (via Put Blob) |
|---|------------------------------------|--|---|
| Version 2019-12-12 and later | 4000 MiB | Approximately 190.7 TiB (4000 MiB X 50,000 blocks) | 5000 MiB (preview) |
| Version 2016-05-31 through version 2019-07-07 | 100 MiB | Approximately 4.75 TiB (100 MiB X 50,000 blocks) | 256 MiB |
| Versions prior to 2016-05-31 | 4 MiB | Approximately 195 GiB (4 MiB X 50,000 blocks) | 64 MiB |

Wielkość partycji w Spark jest konfigurowalna, default = 128MB

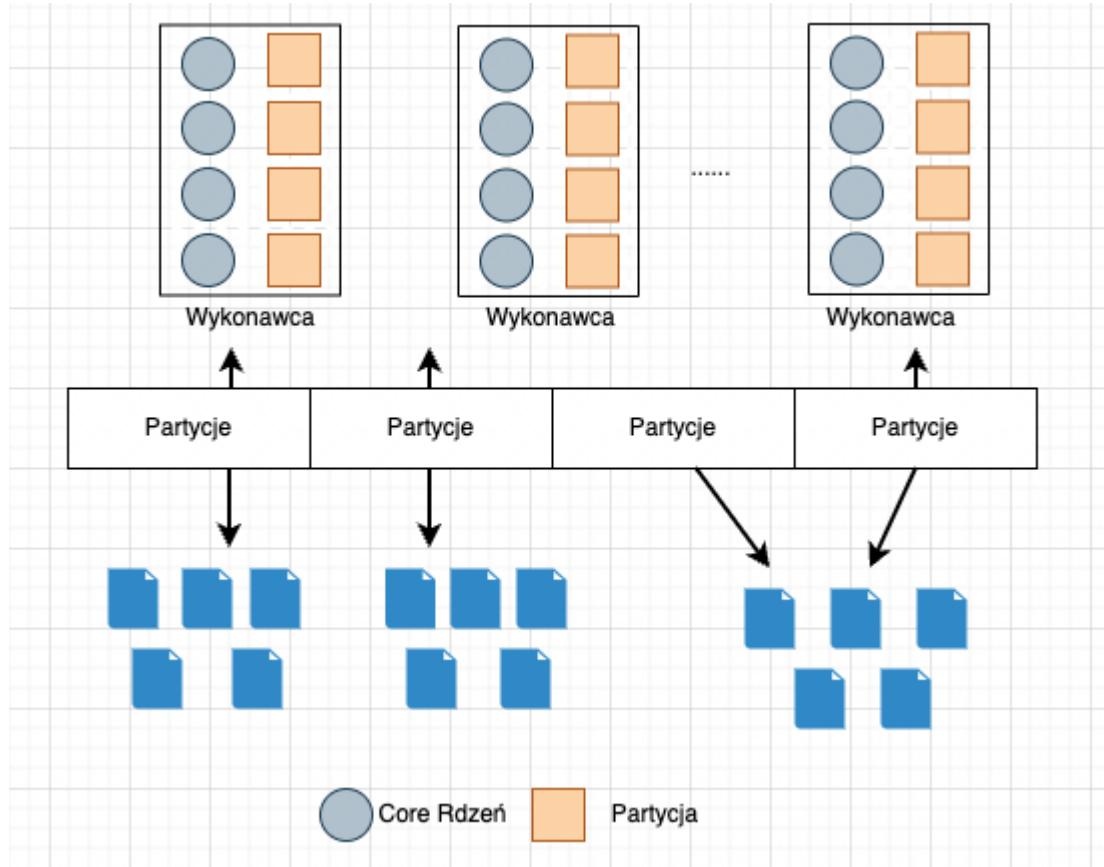
spark.sql.files.maxPartitionBytes.

Jej zmniejszenie może spowodować “**small file problem**”— dużo plików o małych partycjach – I/O. Bardzo to spowolni proces ze względu na konieczność wczytania dużych ilości plików (otwierania, zamykania i “skakania” po folderach).

Partycje Shuffle

Powstają podczas **’shuffle state’ (joins, groupBy)**

Ilość partycji shuffle jest ustawiona na **200** ale możesz to zmienić
spark.sql.shuffle.partitions



Konfiguracja

| Configuration | Default value, recommendation, and description |
|--|---|
| <code>spark.driver.memory</code> | Default is 1g (1 GB). This is the amount of memory allocated to the Spark driver to receive data from executors. This is often changed during <code>spark-submit</code> with <code>--driver-memory</code> . Only change this if you expect the driver to receive large amounts of data back from operations like <code>collect()</code> , or if you run out of driver memory. |
| <code>spark.shuffle.file.buffer</code> | Default is 32 KB. Recommended is 1 MB. This allows Spark to do more buffering before writing final map results to disk. |
| <code>spark.file.transferTo</code> | Default is <code>true</code> . Setting it to <code>false</code> will force Spark to use the file buffer to transfer files before finally writing to disk; this will decrease the I/O activity. |
| <code>spark.shuffle.unsafe.file.output.buffer</code> | Default is 32 KB. This controls the amount of buffering possible when merging files during shuffle operations. In general, large values (e.g., 1 MB) are more appropriate for larger workloads, whereas the default can work for smaller workloads. |
| <code>spark.io.compression.lz4.blockSize</code> | Default is 32 KB. Increase to 512 KB. You can decrease the size of the shuffle file by increasing the compressed size of the block. |
| <code>spark.shuffle.service.index.cache.size</code> | Default is 100m. Cache entries are limited to the specified memory footprint in byte. |
| <code>spark.shuffle.registration.timeout</code> | Default is 5000 ms. Increase to 120000 ms. |
| <code>spark.shuffle.registration.maxAttempts</code> | Default is 3. Increase to 5 if needed. |

Linki

- <https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-DataFrameReader.html>
- [Degrading Performance? You Might be Suffering From the Small Files Syndrome – Databricks](https://databricks.com/knowledge-base/100/degrading-performance-you-might-be-suffering-from-the-small-files-syndrome)
- <https://spark.apache.org/docs/latest/configuration.html>

Zmiana partycji coalesce() vs repartition()

Repartition() zwiększenie bądź zmniejszenie ilości partycji. Wywołuje pełny shuffle więc trzeba mieć to na uwadze.

Coalesce() służy do zmniejszania ilości partycji, i w większości przypadków nie wywoła operacji shuffle. Jeśli chcesz zmniejszyć ilość partycji coalesce będzie bardziej optymalne.

- <https://medium.com/@mrpowers/managing-spark-partitions-with-coalesce-and-repartition-4050c57ad5c4>
- <https://books.japila.pl/apache-spark-internals/rdd/spark-rdd-partitions/?h=repartition#repartitionnumpartitions-intimplicit-ord-orderingt-null-rddt>

Kontrolowanie liczby partycji

Konfiguracja sparka pozwala na kontrolowanie liczby partycji. Jako wartość domyślna ustawiono 200 partycji. Bardzo często nie jest to optymalne. Tą wartość trzeba dobierać indywidualnie w zależności od tego jak działa Twoja aplikacja. Wartość dobierz w zależności od tego ile jest danych i ile jest rdzeni.

Wartości poglądowe:

Dane 200GB (200000MB) / **200MB** = 1000 partycji

Jeśli klatka ma 2000 rdzeni to wtedy 2000 partycji

spark.sql.shuffle.partitions

- <https://medium.com/@manuelmourato25/how-spark-dataframe-shuffling-can-hurt-your-partitioning-28d05fdcb6fa>
- <https://kb.databricks.com/data/random-split-behavior.html>

Optymalizacja

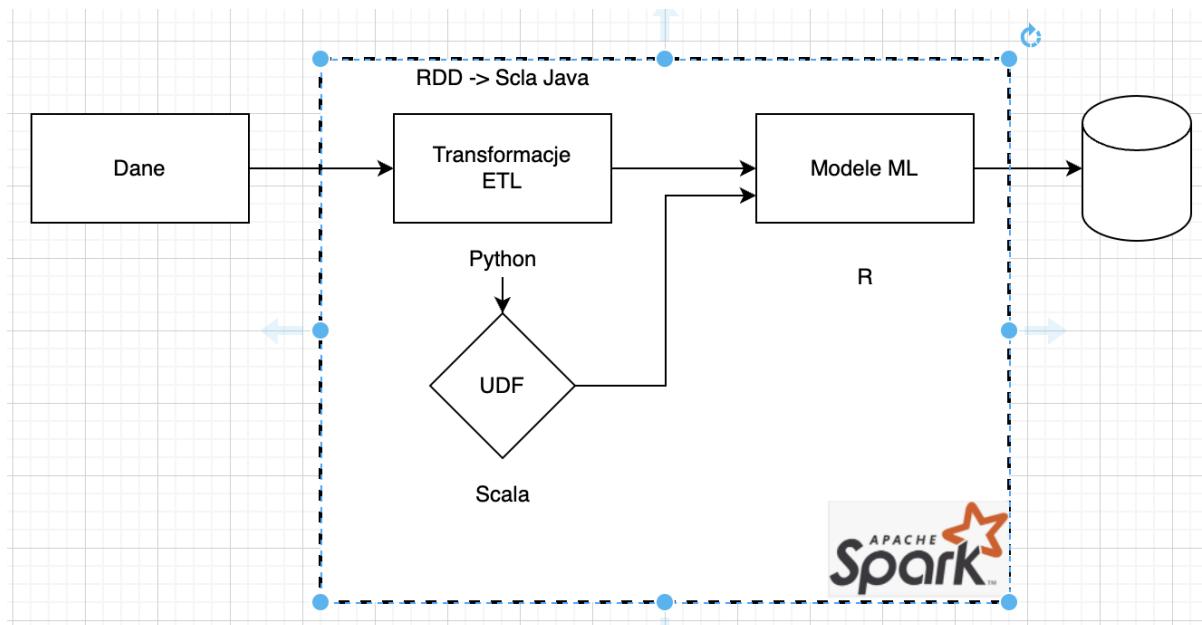
Co można optymalizować

Teoretycznie bardzo wiele, w praktyce możesz się skupić na kilku kluczowych elementach i w praktyce to powinno wystarczyć. Pamiętaj żeby nie popaść w obsesję optymalizacji, bo koszt twojej pracy może znacznie przewyższyć zysk z optymalizacji.

- Decyzje dotyczące kodu (RDD czy DataFrame)
- Łączniki (Joins)
- Agregacja
- Konfiguracje aplikacji
- Dane w spoczynku i w tranzycie
- JVM wykonawcy i driver
- Nodes wykonawcy
- Konfiguracja klastra

Reguły optymalizacji

Dobrze dobierz język, DataFrame czy RDD czy UDFs (Python, R czy Scala)



- **Serializacja obiektów**, użyj **Kryo** jest bardziej efektywne niż Java (**org.apache.spark.serializer.KryoSerializer**) klasy trzeba rejestrować
`spark.kryo.classesToRegister`
`conf.registerKryoClasses(Array(classOf[Class1], classOf[Class2]))`
- **Konfiguracja klastra**:
dynamiczna alokacja w zależności od obciążenia dobra dla wielu aplikacji.

Spark scheduling: poprawia uruchomienie zadań równolegle (`spark.scheduler.mode FAIR`)

- **Składowanie danych:**

Dobierz odpowiedni format, np parquet czy inne pliki binarne (parsing). Użyj odpowiednich plików i kompresji .Zip czy .Tar nie dadzą się dzielić ale gzip, bzip2 tak, ponieważ są stworzone dla środowisk Hadoop i Spark, więc wykorzystaj ich moc.

- **Partycjonowanie tabeli** w wielu ścieżkach opartych na kluczach np. data. Spark przeskoczy ‘skipping’ nieodpowiednie dane.
Uwaga na dużą ilość małych plików - można przedobrzyć jeśli wybierzesz “złą” kolumnę. Zbyt duża liczba partycji będzie spowalniała proces.
- Ilość plików (małe pliki to złe pliki) na szczęście możesz kontrolować liczbę wierszy w pliku **maxRecordsPerFile** a tym samym jego wielkość.
Standardowy blok danych w HDFS ma = 128 MB
Jeśli będziesz miał 30 plików po 5 MB = 30 bloków
 $30 * 5 = 150 \text{ MB}$ (2 bloki)
- **Bucketing** czyli organizacja plików, kontrolujesz które dane są zapisane do konkretnego pliku (możesz uniknąć shuffle). Trzeba jednak bardzo uważać z bucketing bo można narobić więcej złego niż dobrego. Bucketing jest bardzo trudny w zarządzaniu.
Użyj do joins i agg
Unikasz shuffle:
ponieważ dane będą zgrupowane
BucketID => fizyczna partycji
- **Statystyki** dla (cost-base optimizer potrzebuje statystyk)
Statystyki tabel i kolumn - pozwalają na optymalizację przy joinach, agregacjach. Możesz to wywołać (`ANALYZE TABLE COMPUTE STATISTICS / FOR COLUMNS`)
- ‘**Data locality**’ kontrolujesz które nody będą przetrzymywać wybrane dane. (dane nie będą przesyłane przez sieć).
System magazynu danych musi być na tych samych nodach (wsparcie dla ‘locality hints’) ma to HDFS.
- Dobierz ilość partycji do zasobów klastra
- Używaj strukturalnego API (DataFrame) ułatwi GC
<https://spark.apache.org/docs/latest/tuning.html#garbage-collection-tuning>

- Mierz jak często GC wykonuje swoją pracę (spark.executor.extraJavaOptions)
`verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps`

Równoległość

- Rekomendowane 2-3 taski na core przy dużych ilościach danych
`spark.default.parallelism`
`spark.sql.shuffle.partitions`
- Poznaj różnice pomiędzy **repartition** (wywoła shuffle) i **coalesce** (brak shuffle).
- Partycjonowanie przy użyciu RDD
- Unikamy UDFs
- Cache/Persist (pamiętaj o strage level, strona 21)

Skalowanie aplikacji

Dynamiczne alokowanie zasobów dla wielu aplikacji. Szczególnie w streamingu
gdzie mogą wystąpić duże zmiany w obciążeniu.

`spark.dynamicAllocation.enabled true`

`spark.dynamicAllocation.minExecutors 2`

`spark.dynamicAllocation.schedulerBacklogTimeout 1m`

`spark.dynamicAllocation.maxExecutors 20`

`spark.dynamicAllocation.executorIdleTimeout 2min`

Pamięć

Ochrona przed **OOM Errors**, podział jest następujący: **60% wykonawca i 40% magazyn danych**

Różne operacje wymagają innej ilości pamięci (join, agregacja)

`spark.memory.fraction 0.8`

`spark.executor.memory`

Konfiguracja Spark

Zmiana konfiguracji w pliku

```
$SPARK_HOME  
conf/spark-defaults.conf.template,  
conf/log4j.properties.template,  
conf/spark-env.sh.template.
```

Wystarczy zapisać zmiany w pliku bez rozszerzenia .template

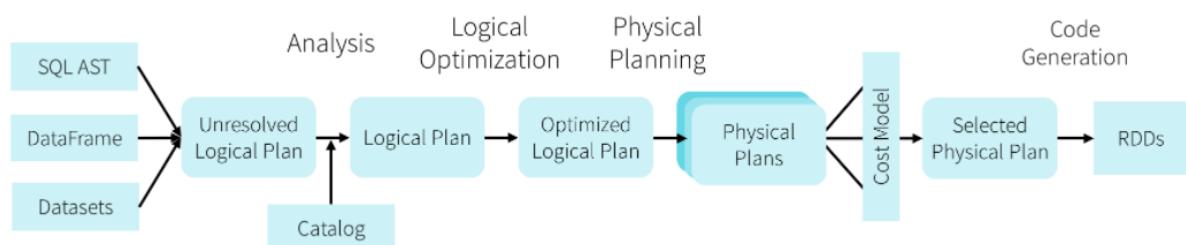
Dodanie konfiguracji podczas wywołania aplikacji

```
spark-submit --conf spark.sql.shuffle.partitions=5 --conf
"spark.executor.memory=2g" --class main.scala.SparkConfig_7_1
jars/mainscala-chapter7_2.12-1.0.jar
```

Optymalizator Catalyst

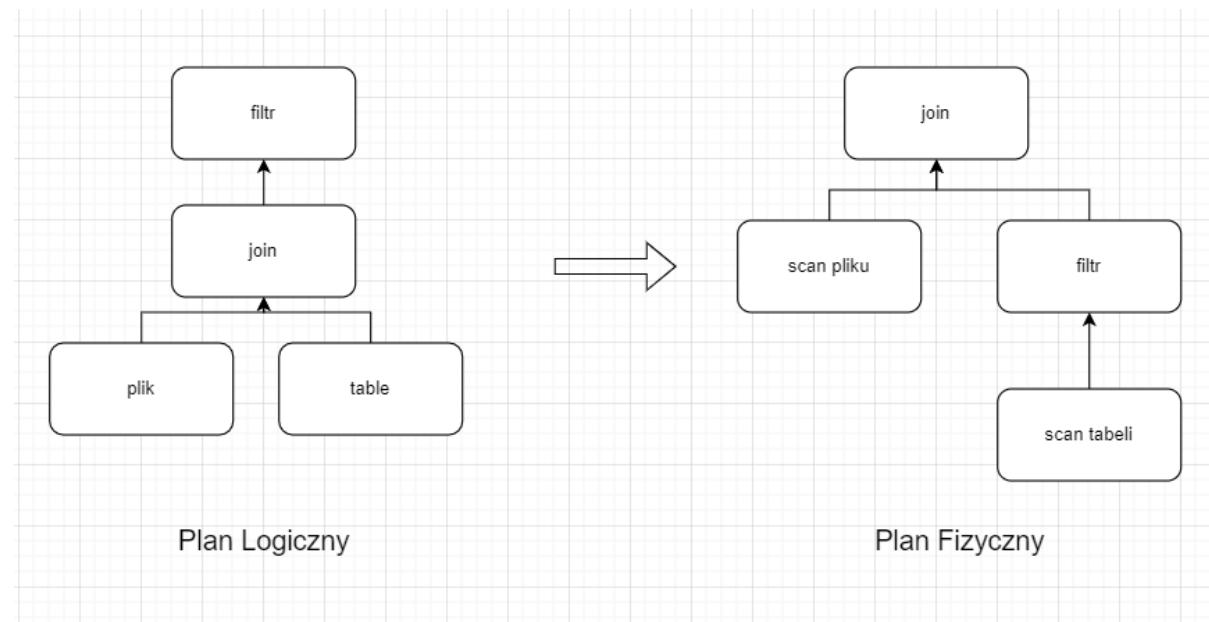
Są cztery główne etapy optymalizatora Catalyst:

1. Analiza
2. Logiczna optymalizacja
3. Fizyczne planowanie
4. Generowanie kodu



Plan Logiczny i Fizyczny

Jak widać na poniższym diagramie fizyczne wykonanie planu różni się od planu logicznego.



Linki

- <https://databricks.com/glossary/catalyst-optimizer>
- [Catalyst Optimizer · The Internals of Spark SQL \(gitbooks.io\)](https://www.gitbooks.io/catalyst-optimizer-the-internals-of-spark-sql/)

Identyfikowanie wąskich gardeł wydajności w aplikacjach Spark

- <https://spark.apache.org/docs/latest/sql-performance-tuning.html>
- <https://databricks.com/session/understanding-the-performance-of-spark-applications>
- <https://databricks.com/blog/2018/01/09/databricks-cache-boots-apache-spark-performance.html>
- <https://databricks.com/session/improving-python-and-spark-performance-and-interoperability>
- <https://databricks.com/training/instructor-led-training/courses/apache-spark-tuning-and-best-practices>
- <https://databricks.com/session/tuning-apache-spark-for-large-scale-workloads>
- [Spark 3.3.0 ScalaDoc - org.apache.spark.storage.StorageLevel](#)

Plan wykonania

W każdej chwili jesteś w stanie sprawdzić jaki jest plan wykonania optymalizatora Catalyst.
Wykorzystaj do tego funkcję **explain()**

Parametry

- **EXTENDED**: pokazuje najwięcej informacji. Generuje trzy plany (parsed, analyzed i optimized). Zoptymalizowany plan logiczny jest przekształcany za pomocą zestawu reguł optymalizacji, czego wynikiem jest plan fizyczny.
- **CODEGEN**: Generuje kod dla instrukcji, jeśli istnieje, oraz plan fizyczny.
- **COST**: Jeśli statystyki węzła są dostępne, generuje plan logiczny oraz jego statystyki.
- **FORMATTED**: Generuje dwie sekcje: zarys planu fizycznego i szczegóły węzła.
- **Statement**: określa instrukcję SQL

cache() alias persist()

cache() nie wykonuje żadnej akcji tylko zaznacza DataFrame jako „**cacheable**”.

Spark zwraca instancję DataFrame - Nie jest to technicznie ani akcja ani transformacja.

Żeby wykonać pełny cache Spark musi przetworzyć każdy element danych co zmaterializuje count().

Cache przetrzymuje dane w RAM wykonawcy.

Persist(StorageLevel.LEVEL) posiada dodatkowe opcje, możesz zadecydować o tym, jak dane są przetrzymywane (StorageLevels). Poniższa tabela wyjaśnia poszczególne opcje . Dane na dysku są zawsze serializowane.

| StorageLevel | Description |
|---------------------|---|
| MEMORY_ONLY | Data is stored directly as objects and stored only in memory. |
| MEMORY_ONLY_SER | Data is serialized as compact byte array representation and stored only in memory. To use it, it has to be deserialized at a cost. |
| MEMORY_AND_DISK | Data is stored directly as objects in memory, but if there's insufficient memory the rest is serialized and stored on disk. |
| DISK_ONLY | Data is serialized and stored on disk. |
| OFF_HEAP | Data is stored off-heap. Off-heap memory is used in Spark for storage and query execution; see "Configuring Spark executors' memory and the shuffle service" on page 178. |
| MEMORY_AND_DISK_SER | Like MEMORY_AND_DISK, but data is serialized when stored in memory. (Data is always serialized when stored on disk.) |

Kiedy użyć Cache i persist

- DataFrames kiedy proces ML wymaga wielokrotnego użycia tych samych danych.
- DataFrames kiedy są wielokrotnie używane w procesie ETL

Kiedy nie używać

- Kiedy DataFrame jest za duży i nie mieści się w pamięci
- Kiedy transformacja jest prosta i nie wymaga użycia danych ponownie w procesie.

Poziomy magazynowania danych

Te poziomy oznaczają gdzie spark ma przetrzymywać dane czy w pamięci czy na dysku (serializowane czy nie).

MEMORY_ONLY,
 MEMORY_AND_DISK,
 MEMORY_ONLY_SER – (serializowany)
 MEMORY_AND_DISK_SER,
 DISK_ONLY, MEMORY_ONLY_2 – (replikuje partycje)
 MEMORY_AND_DISK_2

Linki:

[Spark 3.3.0 ScalaDoc - org.apache.spark.storage.StorageLevel](#)

- Data Serialization
- Memory Tuning
 - Memory Management Overview
 - Determining Memory Consumption
 - Tuning Data Structures
 - Serialized RDD Storage
 - Garbage Collection Tuning

- Other Considerations
 - Level of Parallelism
 - Parallel Listing on Input Paths
 - Memory Usage of Reduce Tasks
 - Broadcasting Large Variables
 - Data Locality
- Summary

Adaptacyjne wykonywanie zapytań

Wykonuje Optymalizację podczas wykonywania zapytań.

Dostosowywanie planów zapytań na podstawie statystyk 'runtime' zbieranych w procesie wykonywania zapytań

Dostępne w wersji Spark 3.0 i Databricks Runtime 7.0, zalety:

Dynamicznie zmienia łączenie sortowania przez łączenie z rozgłoszaniem (sort merge join into broadcast hash join).

Dynamicznie łączy partie (łączy małe partie w partie o "rozsądny rozmiarze") po wymianie losowej (shuffle). Bardzo małe zadania mają gorszą przepustowość IO i są bardziej narażone na obciążenie związane z harmonogramem i konfiguracją zadań.

Dynamicznie obsługuje pochylenie w łączeniu sortowania (sort merge join), scalaniu i mieszaniu połączeń (hash join), dzieląc (i replikując w razie potrzeby) dzieli na zadania o mniej więcej równych rozmiarach.

Dynamicznie wykrywa i propaguje puste relacje.

[How to Speed up SQL Queries with Adaptive Query Execution \(databricks.com\)](#)

Notatnik

[AQE Demo - Databricks](#)

Spark Dataframe

Podstawowe typy danych

<https://spark.apache.org/docs/latest/sql-ref-datatypes.html>

Numeric types

ByteType: Represents 1-byte signed integer numbers. The range of numbers is from -128 to 127.

ShortType: Represents **2-byte** signed integer numbers. The range of numbers is from -32768 to 32767.

IntegerType: Represents **4-byte** signed integer numbers. The range of numbers is from -2147483648 to 2147483647.

LongType: Represents **8-byte** signed integer numbers. The range of numbers is from -9223372036854775808 to 9223372036854775807.

FloatType: Represents 4-byte single-precision floating point numbers. **WARTOŚĆ PRZYBLIŻONA !!!**

DoubleType: Represents **8-byte** double-precision floating point numbers.

DecimalType: Represents arbitrary-precision signed decimal numbers. Backed internally by java.math.BigDecimal. A BigDecimal consists of an arbitrary precision integer unscaled value and a 32-bit integer scale.

WAŻNE:

Dobrze dobieraj typy danych, źle dobrane będą zajmowały dużo miejsca na dysku i w pamięci. Poniższa tabela pokazuje jaki jest rozmiar danych w zależności od wybranego typu.

| Liczba rzędów | Typ | Rozmiar |
|---------------|---------|-----------|
| 100 000 000 | Integer | 190.73 GB |
| 100 000 000 | Long | 381.47 GB |

Konwersja do typów sparka

Konwertuj typy natywne do typów Spark. Robimy to za pomocą funkcji `lit()`

```

1 %sql
2 SELECT 5, "five", 5.0
3

```

▶ (1) Spark Jobs

| | 5 | five | 5.0 |
|---|---|------|-----|
| 1 | 5 | five | 5.0 |

```

6
7 .select( lit( 5 ), lit(" five"), lit( 5.0 ))
8 DataFrame.printSchema

```

▶ `dataFrame: org.apache.spark.sql.DataFrame = [5: integer, five: string ... 1 more field]`

`root`

- |-- `5: integer (nullable = false)`
- |-- `five: string (nullable = false)`
- |-- `5.0: double (nullable = false)`

Czas uniksowy

Czas uniksowy, czas POSIX (ang. **Unix time**, **POSIX time**) – system reprezentacji czasu mierzący liczbę sekund od początku [1970](#) roku [UTC](#), czyli od chwili zwanej początkiem epoki Uniksa (ang. *Unix Epoch*). Nie uwzględnia [sekund przepłynących](#), zatem rzeczywista liczba sekund, jakie upłynęły od początku epoki Uniksa, jest większa o liczbę sekund przepłynących.

```

1 val epoch = System.currentTimeMillis()

```

`epoch: Long = 1646577667482`

Narzędzie do konwersji

<https://www.epochconverter.com/#tools>

Funkcje daty i czasu służące do konwersji StringType to/from DateType or

TimestampType.

Przykłady:

```
unix_timestamp(),
date_format(),
to_unix_timestamp(),
from_unixtime(),
to_date(),
to_timestamp(),
from_utc_timestamp(),
to_utc_timestamp()
```

Linki

- <https://spark.apache.org/docs/3.2.1/sql-ref-datetime-pattern.html>
- <https://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html>
- A Comprehensive Look at Dates and Timestamps in Apache Spark™ 3.0
<https://databricks.com/blog/2020/07/22/a-comprehensive-look-at-dates-and-timestamps-in-apache-spark-3-0.html>

Skomplikowane Typy Danych

| Data type | Scala | API |
|---------------|--|--|
| BinaryType | Array[Byte] | DataTypes.BinaryType |
| TimestampType | java.sql.Timestamp | DataTypes.TimestampType |
| DateType | java.sql.Date | DataTypes.DateType |
| ArrayType | scala.collection.Seq | DataTypes.createArrayType(ElementType) |
| MapType | scala.collection.Map | DataTypes.createMapType(keyType,valueType) |
| StructType | org.apache.spark.sql.Row | StructType(ArrayType[fieldTypes]) |
| StructField | A value type corresponding to the type of this field | StructField(name, dataType, [nullable]) |

ArrayType-MapType-StructType

```
val maps = StructType(
  StructField("longs2strings", createMapType(LongType, StringType), false) :: Nil)

scala> maps.prettyJson
res0: String =
{
  "type" : "struct",
  "fields" : [ {
    "name" : "longs2strings",
    "type" : {
      "type" : "map",
      "keyType" : "long",
      "valueType" : "string",
      "valueContainsNull" : true
    },
    "nullable" : false,
    "metadata" : { }
  } ]
}
```

```
import org.apache.spark.sql.types.DataTypes

scala> val arrayType = DataTypes.createArrayType(BooleanType)
arrayType: org.apache.spark.sql.types.ArrayType = ArrayType(BooleanType,true)

scala> val mapType = DataTypes.createMapType(StringType, LongType)
mapType: org.apache.spark.sql.types.MapType = MapType(StringType,LongType,true)
```

Linki

- <https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-Data-Type.html>
- <https://docs.microsoft.com/bs-latn-ba/azure/databricks/delta/data-transformation/complex-types>

Schematy Danych

```

1 import org.apache.spark.sql.types.{IntegerType, StringType, StructType, StructField}
2
3 val schemat = StructType(Array(
4   StructField("author", StringType, false),
5   StructField("title", StringType, false),
6   StructField("pages", IntegerType, false)))
7
8

```

```

1 val schemat = "author STRING, title STRING, pages INT"

```

```

1 import java.io.File
2 import java.nio.file.{Paths, Files}
3 import java.nio.charset.StandardCharsets
4 import org.apache.spark.sql.types._
5
6 val schemaFileName = Paths.get("/dbfs/mnt/source/shows-schema.json")
7 val schemaContent = new String(Files.readAllBytes(schemaFileName), StandardCharsets.UTF_8)
8 val schema = DataType.fromJson(schemaContent).asInstanceOf[StructType]

```

Kiedy używamy schematu

Podczas odczytu

`DataFrameReader.format(...).option("key",
"value").schema(...).load()`

```

1
2
3 val file = spark.read.format("csv")
4   .option("header","true")
5   .option("inferSchema","true")
6   .load("dbfs:/dbfs/FileStore/Tables/Files/actors.csv/")
7

```

Podczas zapisu

`DataFrameWriter.format(...).option(...).
partitionBy(...).bucketBy(...).sortBy(...).save()`

```
3 val file = spark.read.format("csv")
4 .option("header","true")
5 .schema(schematPliku)
6 .load("dbfs:/dbfs/FileStore/Tables/Files/actors.csv/")
7
```

Linki

<https://spark.apache.org/docs/latest/sql-data-sources-json.html>

DataFrame i zestawy danych

Są to kolekcje ustrukturyzowane reprezentujące zbiory tabelaryczne z wierszami i kolumnami. Są niezmienne, co oznacza, że nie możesz ich zmienić po utworzeniu. Schemat („schema”) wymusza nazwy kolumn i typy danych, takie jak ciąg lub liczba całkowita. Typy Spark są mapowane bezpośrednio na interfejsy API różnych języków.

Bezpieczne ładowanie danych

- CSV
- JSON
- Parquet
- ORC
- Połączenia JDBC / ODBC
- Zwykłe pliki tekstowe

W pracy z danymi ważne jest, żeby złe dane nie przedostały się ze źródła do celu. I tutaj Spark daje Ci kilka opcji. Są to tzw ‘**Read Modes**’, czyli opcje jakie masz podczas odczytu danych.

Typy

- **PERMISSIVE**: Jeśli atrybuty nie mogą zostać wczytane Spark zamienia je na nule
- **DROPMALFORMED**: wiersze są usuwane
- **FAILFAST**: proces odczytu zostaje całkowicie zatrzymany

Na produkcji powinieneś przekierować dane do osobnej ścieżki, gdzie zbudujesz osobny mechanizm przeładowania błędnych danych.

```
val filewithschema = spark.read.format("json")
.option("schema", schema)
.option("badRecordsPath", "/mnt/source/badrecords")
.load("/mnt/source/shows.json")
```

Tryby odczytu (Read Modes)

```

1 // Opcja 1
2 val filewithschema = spark.read.format("json")
3 .option("schema",schema)
4 .option("badRecordsPath", "/mnt/source/badrecords")
5 .load("/mnt/source/shows.json")
6
7 //Opcja 2
8 val filewithschema = spark.read.format("json")
9 .option("schema",schema)
10 .option("mode", "PERMISSIVE")
11 .load("/mnt/source/shows.json")
12
13 //Opcja 3
14 val filewithschema = spark.read.format("json")
15 .option("schema",schema)
16 .option("mode", "DROPMALFORMED")
17 .load("/mnt/source/shows.json")
18
19 // Opcja 4
20 val filewithschema = spark.read.format("json")
21 .option("schema",schema)
22 .option("mode", "FAILFAST")
23 .load("/mnt/source/shows.json")

```

Tryby Zapisu (Save Modes)

| Scala/Java | Any Language | Meaning |
|-------------------------------------|--|--|
| SaveMode.ErrorIfExists (default) | "error" or "errorIfExists" (default) | When saving a DataFrame to a data source, if data already exists, an exception is expected to be thrown. |
| SaveMode.Append | "append" | When saving a DataFrame to a data source, if data/table already exists, contents of the DataFrame are expected to be appended to existing data. |
| SaveMode.Overwrite | "overwrite" | Overwrite mode means that when saving a DataFrame to a data source, if data/table already exists, existing data is expected to be overwritten by the contents of the DataFrame. |
| SaveMode.Ignore | "ignore" | Ignore mode means that when saving a DataFrame to a data source, if data already exists, the save operation is expected not to save the contents of the DataFrame and not to change the existing data. This is similar to a CREATE TABLE IF NOT EXISTS in SQL. |

Agregacie

Kiedy coś agregujesz, ostatecznie zbierasz podsumowania danych. Jest to fundament analityki Big Data. W agregacji określisz klucz lub grupę i funkcję agregacji.

Proste grupowanie („Simple grouping”): podsumowanie DataFrame poprzez wykonanie agregacji w instrukcji select.

Grupuj według klucza („Group by”): określ jeden lub więcej kluczy i jedną lub więcej agregacji, aby przekształcić kolumny wartości.

Okno („Window”): jest podobne do grupy, ale wiersze wprowadzone do funkcji są w jakiś sposób powiązane z bieżącym wierszem.

Zestaw grupujący („Grouping set”): można stosować agregację na wielu różnych poziomach. Są one dostępne jako operacje podstawowe w SQL oraz poprzez pakiety zbiorcze i kostki w DataFrames.

Zestawienie („Rollup”): określ jeden lub więcej kluczy i jedną lub więcej agregacji, aby przekształcić kolumny wartości, które zostaną podsumowane hierarchicznie.

Kostka („Cube”): określ jeden lub więcej kluczy i jedną lub więcej agregacji, aby przekształcić kolumny wartości, które zostaną podsumowane we wszystkich kombinacjach kolumn.

DataSet

Podstawy Strukturalnego API

- DataFrame to **DataSet** typu **Row**
- Działają w JVM czyli tylko Java I Scala
- Możesz zdefiniować obiekt dla każdego wiersza danych w DataSet. Mogą być przetworzone równolegle przy użyciu funkcyjnych lub relacyjnych operacji np. map(), reduce(), filter(), select(), aggregate()
- Obiekty silnie typowane (Strongly typed)
- Nie ma możliwości wczytania schematu, trzeba go stworzyć wcześniej. Typy są automatycznie mapowane zgodnie z definicją w case class.
- DataFrame w Scali jako alias dla kolekcji generycznego obiektu **Dataset[Row]**, Row jest ogólnym beztypowym obiektem JVM, w którym możesz trzymać różne typy pól.
- Dataset (strongly types) obiekty JVM w Scali lub klasa w Javie. Każdy zbiór danych [w Scali] ma również (untyped view) widok o nazwie DataFrame, który jest zestawem danych wiersza.

DataFrame – Python, R (typy domyślne podczas wykonania)

Dataset – Scala, Java (typy podczas komplikacji)

Encoders

Spark ma wewnętrzne typy danych.

StringType, BinaryType, IntegerType, BooleanType, MapType

Encoders są odpowiedzialne za mapowanie typów dla danego języka podczas operacji Sparda.

Kiedy użyć Datasets

- Kiedy operacje nie mogą być przetworzone przy użyciu DataFrame.
- Kiedy masz specyficzną logikę biznesową, którą chcesz zakodować w specjalną funkcję.
- Kiedy zależy ci na “type-safety”, i jesteś w stanie zaakceptować koszt związany z serializacją: Dataset API is type-safe. Operacje, która nie spełniają walidacji typów np odejmowanie dwóch stringów – wyjątek pojawi się podczas komplikacji a nie uruchomienia.
- Kiedy chcesz pobrać DataFrames do lokalnego dysku, będą one w wybranym typie klasy co ułatwi manipulacje.
- Jednym z popularniejszych scenariuszy jest użycie obu DataFrames i Datasets równocześnie, decydując kiedy będzie lepszy performance a kiedy type-safety.

Tworzenie Dataset

W celu stworzenia Dataset musisz stworzyć case class. Jej główne zalety:

1. Niezmienna (Immutable)
2. Wsparcie dla “pattern matching”
3. Można porównać jej strukturę a nie odniesienie
4. Łatwa w manipulacji

```
case class IoTData(level: Long, start: Long, timestamp: Timestamp, scale: String)

val dataSet = spark.read.format("json").load("dbfs/tmp/").as[IoTData]
```

Transformacje

- Transformacje w zestawach danych są takie same, jak te, w DataFrames.
- Oprócz tych przekształceń zestawy danych umożliwiają nam określenie bardziej złożonych i silnie typowanych transformacji, niż mógłbyś wykonać na ramkach DataFrames.
- Manipuluj surowymi typami wirtualnej maszyny Java (JVM).

Łączniki

W DataSet są dodatkowe metody np joinWith () – tworzy zagnieżdżone sety, każda kolumna reprezentuje jeden Dataset. Przydatne w tworzeniu zaawansowanych filtracji czy map.

Możesz użyć normalnego Joina (zwróci DataFrame).

Grupowanie i agregację

- Agregacje podlegają tym samym standardom, te same funkcje będą działać co na DataFrame (groupBy, rollup, cube) ale zwrócią DataFrame (tracisz Dataset).
- Jeśli chcesz otrzymać Dataset to użyj groupByKey(funkcja) - **uwaga** taka funkcja nie może być zoptymalizowana przez Spark.
- Grupowanie z kluczem daje Ci możliwość wykonania funkcji na poszczególnych grupach (Klucz Wartość).

Błędy komplikacji i wykonania

W zależności czego użyjesz błędy będą się pojawiać podczas uruchomienia kodu lub podczas komplikacji.

| Błąd | SQL | DataFrame | Dataset |
|---------|---------------------|---------------------|------------|
| Syntax | Uruchomie (runtime) | Kompilacja | Kompilacja |
| Analiza | Uruchomie (runtime) | Uruchomie (runtime) | Kompilacja |

Linki

<https://docs.scala-lang.org/tour/case-classes.html>

Kiedy użyć DataFrame a kiedy Dataset

- Kiedy chcesz żeby Spark robił **co mu każesz** a nie **jak**, użyj DataFrames lub Datasets.
- Jeśli chcesz bezpieczeństwa, żeby błędy były wychwycone podczas komplikacji (compile-time type safety) nie masz nic przeciwko tworzeniu case classes Dataset[T], use Datasets.
- Dla wyrażeń, filtrów, maps, agregacji, sum, SQL, dostęp do kolumn - DataFrames.
- Dla relacyjnych transformacji SQL-like - DataFrames.
- Jeśli chcesz wykorzystać serializację Tungsten's (Encoders) użyj Datasets.
- Dla ułatwienia i optymalizacji - DataFrames.
- Dla R lub Python użyj DataFrames.

Spark SQL

- Działa jako OLAP a nie OLTP
- Ujednolica komponenty Sparka i zezwala na pracę w DataFrames/Datasets Java, Scala, Python, and R.
- Łączy się z Apache Hive metastore.
- Odczytuje i zapisuje ustrukturyzowane dane z określonym schematem z ustrukturyzowanych formatów plików (JSON, CSV, Text, Avro, Parquet, ORC, etc.)
- Konwertuje dane do tymczasowych tabel.
- Oferuje interaktywny Spark SQL shell do eksploracji danych.
- Dostęp do standardowych konektorów JDBC/ ODBC.
- Generuje zoptymalizowane plany zapytań i kompaktowy kod dla JVM do ostatecznego wykonania.

Dzięki Spark SQL możesz uruchamiać zapytania SQL względem widoków lub tabel. Spark zapewnia funkcje do analizy planów zapytań w celu optymalizacji obliczeń. Zapewnia to integrację bezpośrednio z DataFrames i Datasets API.

Spark SQL jest w stanie wykorzystać w dowolnym przepływie danych. Jego interfejs API pozwala na ekstrakcję danych za pomocą SQL zmanipulowanego jako DataFrame.

Kolejną dobrą wiadomością jest to, że Spark SQL ma dobre relacje z Hive. Hive przechowuje informacje o tabelach do użytku między sesjami w „metastore”. Jest to przydatne dla użytkowników pracujących w niektórych starszych systemach Hadoop.

Spark SQL CLI

Spark SQL CLI to wygodne narzędzie, za pomocą którego można wykonywać podstawowe zapytania Spark SQL w trybie lokalnym. Wszystko to jest dostępne w wierszu poleceń.

Aby uruchomić interfejs Spark SQL CLI, uruchom następujące polecenie w katalogu Spark: ./bin/spark-sql

Interfejs Spark SQL

```
spark.sql(“SELECT 1 + 1”).show()
```

SQL Połączenia

Spark zapewnia interfejs Java Database Connectivity (JDBC), za pomocą którego Ty lub program zdalny łączycie się ze sterownikiem Spark w celu wykonania Spark SQL

Append to SQL Table

Python

```
try:  
    df.write \  
        .format("com.microsoft.sqlserver.jdbc.spark") \  
        .mode("append") \  
        .option("url", url) \  
        .option("dbtable", table_name) \  
        .option("user", username) \  
        .option("password", password) \  
        .save()  
except ValueError as error :  
    print("Connector write failed", error)
```

Read from SQL Table

Python

```
jdbcDF = spark.read \  
    .format("com.microsoft.sqlserver.jdbc.spark") \  
    .option("url", url) \  
    .option("dbtable", table_name) \  
    .option("user", username) \  
    .option("password", password).load()
```

Otwarte na narzędzia Business Intelligence, takie jak PowerBI

Links

<https://docs.microsoft.com/en-us/sql/connect/spark/connector?view=sql-server-ver15>

Tabele

Ramki DataFrames można również zapisywać jako tabele w magazynie metadanych Hive za pomocą polecenia **saveAsTable**. Zwróć uwagę, że istniejące wdrożenie Hive nie jest konieczne do korzystania z tej funkcji. Spark utworzy domyślny lokalny metastor Hive. W przeciwieństwie do polecenia **createOrReplaceTempView**, **saveAsTable** zmaterializuje zawartość DataFrame i utworzy wskaźnik do danych w magazynie metadanych Hive.

```
1 df.write.option("path", "/some/path").saveAsTable("t")
```

```
1 dataFrame.createOrReplaceTempView("Tymczasowa_Tabela")
```

SQL Interfejs

```
1 val sql = spark.sql("SELECT * FROM Tymczasowa_Tabela")
```

Widoki

Widok określa zestaw transformacji istniejącej tabeli — po prostu zapisane plany zapytań, które mogą być wykorzystane do ponownego użycia logiki przy wykonywaniu zapytań.

```
CREATE VIEW City AS  
|   SELECT * FROM tabela WHERE city_code = 'krk'
```

Widoki tymczasowe (Temporary views) są aktywne podczas jednej sesji. Kiedy sesja się skończy zostaną automatycznie usunięte.

```
CREATE TEMP VIEW Country AS  
|   SELECT * FROM tabela WHERE country_code = 'krk'
```

Widoki globalne (Global temporary views) są zapisane w bazie global_temp.

```
CREATE GLOBAL TEMP VIEW Country AS  
|   SELECT * FROM tabela WHERE country_code = 'krk'
```

Szablon

SQL

```
CREATE [OR REPLACE] [[GLOBAL] TEMPORARY] VIEW [db_name.]view_name
[(col_name1 [COMMENT col_comment1], ...)]
[COMMENT table_comment]
[TBLPROPERTIES (key1=val1, key2=val2, ...)]
AS select_statement
```

Spark API

```
dataFrame.createGlobalTempView("tabelaGlobal")
dataFrame.createOrReplaceTempView("tabelaTemp")

val tabelaGlobal = spark.sql("Select * from global_temp.tabelaGlobal").show()
val tabelaTemp = spark.sql("Select * from tabelaTemp").show()
```

Linki

- <https://spark.apache.org/docs/latest/sql-ref-syntax-aux-show-views.html>
- <https://docs.microsoft.com/pl-pl/azure/databricks/spark/2.x/spark-sql/language-manual/create-view>
- [SQL Reference - Spark 3.3.0 Documentation \(apache.org\)](https://spark.apache.org/docs/3.3.0/api/python/reference/pyspark.sql.html#pyspark.sql.SQLContext.createGlobalTempView)
- [Zewnętrzny magazyn metadanych Apache Hive — Azure Databricks | Microsoft Learn](https://learn.microsoft.com/pl-pl/azure/databricks/metastore/apache-hive-metastore)

Apache Hive

Hive to system hurtowni danych ułatwiający odczytywanie, zapisywanie i zarządzanie dużymi zestawami danych przechowywanymi w rozproszonej pamięci masowej za pomocą SQL.

- ❖ Strukturę można narzucić na istniejące dane. Masz dostępne narzędzia tj. komendy wiersza poleceń oraz sterowniki JDBC do łączenia użytkowników z Hive.
- ❖ Spark SQL ma świetne relacje z Hive, ponieważ może łączyć się z Hive ‘metastore’. Magazyn metadanych Hive to sposób, w jaki Hive przechowuje informacje o tabelach i do użytku między sesjami.
- ❖ API pozwala na wyodrębnianie danych za pomocą SQL, manipulowanie jako DataFrame, przekazywanie do jednego z algorytmów uczenia maszynowego

Spark MLlib na dużą skalę, zapisywanie do innego źródła danych i wszystkiego pomiędzy.

- ❖ Spark SQL jest całkowicie zgodny z instrukcjami Hive SQL (HiveQL). Obsługuje schematem przy odczycie i transparentnie konwertuje zapytania do Apache Spark, MapReduce.
- ❖ Rozproszona pamięć masowa, która jest częścią Apache Hadoop.
- ❖ Hive może być używany lokalnie jak i w chmurze z użyciem różnymi nośników. Azure Cloud Storage, AWS S3 and Google Cloud Storage, HDFS

Ustawienia Hive

| Property Name | Meaning |
|---|--|
| fileFormat | A fileFormat is kind of a package of storage format specifications, including "serde", "input format" and "output format". Currently we support 6 fileFormats: 'sequencefile', 'rcfile', 'orc', 'parquet', 'textfile' and 'avro'. |
| inputFormat, outputFormat | These 2 options specify the name of a corresponding InputFormat and OutputFormat class as a string literal, e.g. org.apache.hadoop.hive.ql.io.orc.OrcInputFormat. These 2 options must be appeared in a pair, and you can not specify them if you already specified the fileFormat option. |
| serde | This option specifies the name of a serde class. When the fileFormat option is specified, do not specify this option if the given fileFormat already include the information of serde. Currently "sequencefile", "textfile" and "rcfile" don't include the serde information and you can use this option with these 3 fileFormats. |
| fieldDelim, escapeDelim, collectionDelim, mapkeyDelim, lineDelim | These options can only be used with "textfile" fileFormat. They define how to read delimited files into rows. |

Hive wymaga dużej liczby bibliotek, które nie są włączone w standardową dystrybucję Sparka.

Classpath – jeśli Spark zobaczy biblioteki Hive to je załaduje automatycznie, wymagane są do serializacji i deserializacji.

```
// warehouseLocation points to the default location for managed databases and tables
val warehouseLocation = new File("spark-warehouse").getAbsolutePath

val spark = SparkSession
  .builder()
  .appName("Spark Hive Example")
  .config("spark.sql.warehouse.dir", warehouseLocation)
  .enableHiveSupport()
  .getOrCreate()
```

Linki

- <https://docs.microsoft.com/pl-pl/azure/databricks/spark/latest/spark-sql/language-manual/sql-ref-syntax-ddl-create-table-hiveformat>
- <https://docs.microsoft.com/pl-pl/azure/databricks/data/data-sources/hive-tables>
- <https://spark.apache.org/docs/latest/sql-data-sources-hive-tables.html>

Katalog

Katalog to abstrakcja przechowywania metadanych dotyczących danych przechowywanych w tabelach, a także innych przydatnych rzeczy, takich jak bazy danych, tabele, funkcje i widoki.

Katalog jest dostępny w pakiecie **org.apache.spark.sql.catalog.Catalog** i zawiera szereg pomocnych funkcji.

Linki

<https://spark.apache.org/docs/latest/api/scala/org/apache/spark/sql/catalog/Catalog.html>

Odczyt i zapis DataFrames

DataFrame czyli podstawowy element struktury danych, jaki daje nam API. Możesz ją stworzyć podczas wczytywania pliku z dysku lub użyć jakiejś kolekcji.

```
filePath = "dbfs:/FileStore/tables/Files/file.csv"
actorsDf = spark.read.format("csv") \
  .option("header","true") \
  .option("inferSchema","true") \
  .load(filePath)
```

Dane możesz pobierać np. z relacyjnej bazy danych

```
val jdbcHostname = "jakisserver.database.windows.net"
```

```

val jdbcPort = 1433
val jdbcDatabase = "testdb"

val tabela = spark.read
  .format("jdbc")
  .option("url",
    s"jdbc:sqlserver://${jdbcHostname}:${jdbcPort};database=${jdbcDatabase}")
  .option("user", "admin")
  .option("password", "Password123$")
  .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver")
  .option("query", "SELECT * FROM INFORMATION_SCHEMA.TABLES")
  .load()

```

Operacje na kolumnach

Wybieranie, zmiana nazwy, sortowanie, filtrowanie, zmiana typów danych,

Klasa 'Column' daje nam więcej opcji.

Oto lista popularnych funkcji

Funkcje Column

- Różne funkcje matematyczne add, subtract, multiply & divide
- Sprawdzanie wartości null `isNull()`, `isNotNull()` & `isNaN()`.
- Zwracanie aliasów kolumny `alias()` & `name()`.
- `between()` - Wyrażenie zwraca boolean jeżeli wartość kolumny jest pomiędzy wybranymi wartościami.
- `cast()` & `astype()` - konwersja kolumny do innego typu danych.
- `asc()` - Wyrażenie rosnącego sortowania dla wybranej kolumny.
- `desc()` - Wyrażenie malejącego sortowania dla wybranej kolumny.
- `startswith()` - Wartość kolumny zaczyna się od.
- `endswith()` - Wartość kolumny kończy się na.
- `isin()` - Wyrażenie logiczne, które jest oceniane jako prawda, jeśli wartość tego wyrażenia jest zawarta w wartościach argumentów.
- `like()` - Wyrażenia, które działają jak SQL like.

- `substr()` - Wyrażenia, które zwraca część string.

```
# 1. Wybieramy kolumnę z naszego DataFrame
from pyspark.sql.functions import *
columnName = namesDf.select(col("name"))

# between() - Wyrażenie zwraca boolean jeżeli wartość kolumny jest pomiędzy wybranymi
wartosciami.
betweenDf = ratingsDf.select("total_votes", col("total_votes").between(100,1000))

# Sprawdzamy typy danych przed zmiana
ratingsDf.select(col("total_votes")).printSchema

# startswith() - Wartość kolumny zaczyna się od. lub endwith() czyli konczy się na...
startDf = namesDf.select("name").where(col("name").startswith("Adam")).orderBy("name")

# like() - Wyrażenia, które działa jak SQL like.
likeDf = moviesDf.select("original_title", col("original_title").like("%the%"))

# `substr()` - Wyrażenia, które zwraca część string.
substringDf = moviesDf.select("original_title", col("original_title").substr(1,5))

# Sortowanie, sortujemy po kolumnie 'name'
from pyspark.sql.functions import col
sortedDf = namesDf.orderBy(col("name").asc())
```

Zarządzanie nulls

Istnieje wiele problemów, jeśli chodzi o wartości null i liczenie. Na przykład podczas wykonywania `count(*)`, **Spark zliczy wartości null (w tym wiersze zawierające wszystkie wartości null)**. Jednak podczas liczenia pojedynczej kolumny Spark nie zliczy wartości null.

Spark automatycznie wczyta i rozpozna brak wartości przy tworzeniu Dataframe. Dobra wiadomość jest taka, że spark świetnie sobie poradzi z null. Pamiętaj o tym, że możesz mieć problemy z null jeśli zaczniesz stosować UDFs. Tutaj reguły będą inne w zależności jakiego języka używasz.

```
# Sprawdzanie wartości null isNull(), pozostałe funkcje możesz sprawdzić w ten sam sposób
from pyspark.sql.functions import *
isnullHeight = namesDf.withColumn("is_height_null", col("height").isNull()) \
.withColumn("is_height_not_null", col("height").isNotNull())
```

Linki:

- [Using the Scala Option, Some, and None idiom \(instead of Java null\) | alvinalexander.com](#)
- [NULL Semantics](#)

Transformacje i akcje

Transformacje to operacje, które nie zostaną wykonane w momencie ich napisania/uruchomienia - zostaną uruchomione dopiero gdy wywołasz **akcję**.

Przykładową transformacją może być zmiana typu danych int na float albo przefiltrowanie zbioru danych. Lista przykładowych transformacji w tabeli poniżej. Przetwarzają jeden DataFrame w drugi bez modyfikacji oryginalnego.

Operacja nie jest wykonywana natychmiast tylko zapisana jako lineage (historia operacji).

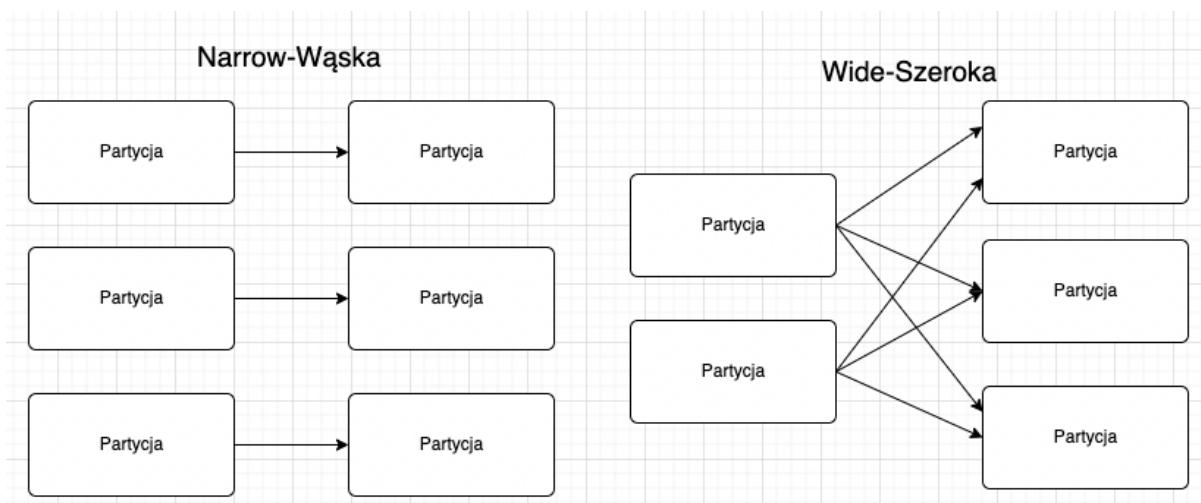
To pozwala na późniejszą optymalizację planu wykonania (execution plan).



Transformacje przykłady

| Transformation | Meaning |
|---|---|
| <code>map(func)</code> | Zwraca rozproszony zestaw danych utworzony przez przekazanie każdego elementu danych do funkcji. |
| <code>filter(func)</code> | Zwraca nowy zestaw danych utworzony przez wybranie tych elementów, dla których funkcja zwróciła wartość true. |
| <code>flatMap(func)</code> | Podobny do funkcji (map), z wyjątkiem tego, że każdy element wejściowy można mapować na jeden lub więcej elementów wyjściowych. Więc funkcja może zwrócić sekwencje. |
| <code>reduceByKey(func, [numPartitions])</code> | Jeśli uruchamiasz funkcję na zestawie danych w formacie (K:klucz, V: Wartość) i funkcja zwraca zestaw danych w formacie (K:klucz, V: Wartość). Kiedy wartość każdego klucza jest agregowana przy użyciu funkcji redukującej, która musi być typem (V,V) => V. Można ustawić liczbę zadań redukujących przy użyciu drugiego parametru. |
| <code>aggregateByKey(zeroValue) (seqOp, combOp, [numPartitions])</code> | Jeśli uruchamiasz funkcję na zestawie danych w formacie (K:klucz, V: Wartość) i funkcja zwraca zestaw danych w formacie (K, U), gdzie wartości są agregowane przy użyciu łączonych funkcji i neutralnej wartości "zero". Pozwala to na agregowanie typów, które są inne niż typy wejściowe. Tak jak w przypadku 'groupByKey', Można ustawić liczbę zadań redukujących przy użyciu drugiego parametru. |
| <code>sortByKey([ascending], [numPartitions])</code> | Jeśli uruchamiasz funkcję na zestawie danych w formacie (K:Klucz, V: Wartość), gdzie K implementuje sortowanie i zwraca posortowany zestaw danych rosnąco lub malejąco. |
| <code>coalesce(numPartitions)</code> | Zmniejsza ilość partycji w RDD do zdefiniowanej liczby. Użyteczne, kiedy używasz filtra na dużym zestawie danych, pozwala uniknąć pustych partycji. dataset |
| <code>repartition(numPartitions)</code> | Mожет быть использована для изменения количества партиций RDD. Необходимо помнить, что функция выполняет операцию 'shuffle' или пересыпает данные через сеть, что является дорогостоящим действием. |

Transformacje Wide – Narrow



Akcje

Akcje to komendy, których wynik jest obliczany w chwili ich uruchomienia. Takie uruchomienie obejmuje też wykonanie wszystkich transformacji, które poprzedzały akcję. Akcja składa się z jednego lub więcej zadań, które zostaną wykonane równolegle przez wiele wykonawców.

Akcje przykłady

| Action | Meaning |
|---|--|
| <code>reduce(func)</code> | Agreguje elementy zestawu danych używając funkcji, która pobiera dwa parametry i zwraca jeden. |
| <code>collect()</code> | Zwraca wszystkie elementy zestawu danych jako tablicę. Wszystkie dane są zbierane na sterowniku. Przy dużej ilości danych może to być problemem i spowodować niestabilność sterownika. |
| <code>count()</code> | Zwraca liczbę elementów zjadających się w zestawie danych. |
| <code>first()</code> | Zwraca pierwszy wiersz zestawu danych. |
| <code>take(n)</code> | Zwraca tablicę z n ilością elementów w zestawie danych. |
| <code>takeSample(withReplacement, num, [seed])</code> | Zwraca tablicę z losową liczbą elementów z zestawu danych. Można użyć zamiennika i generatorem liczb losowych. |
| <code>foreach(func)</code> | Uruchamia funkcję na każdym elemencie zestawu danych. |

Linki

- <https://docs.databricks.com/spark/latest/dataframes-datasets/introduction-to-dataframes-scala.html>
- <https://spark.apache.org/docs/2.2.0/sql-programming-guide.html>
- <https://spark.apache.org/docs/latest/sql-data-sources-load-save-functions.html>
- <https://databricks.com/glossary/what-are-transformations>
- <https://supergloo.com/spark-scala/apache-spark-examples-of-transformations/>

Shuffle

Shuffle – wymiana danych pomiędzy partycjami w klastrze + zapis na dysk

Przykład: grupowanie groupBy(„Nazwisko”)

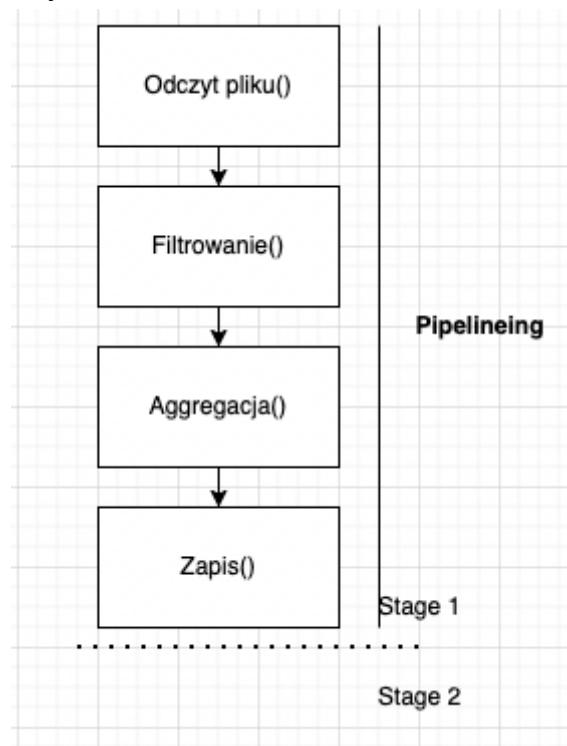
Spark Proces

1. Konwersja danych Tungsten Binary Format.
2. Zapis danych na dysku
3. Wysłanie danych do kolejnego wykonawcy
4. Spark sterownik przydziela, który wykonawca dostaje dane.
5. Wykonawca wyciąga dane od innego wykonawcy (shuffle files).
6. Kopiowanie danych do RAM kolejnego wykonawcy

Pipelining

Jest to metoda grupowania operacji, Spark stara się uruchomić tyle operacji ile się da na pojedynczej partycji.

Kiedy pojedyncza partycja jest wczytana do RAMu, Spark stara się wykonać jak najwięcej operacji w jednym Tasku.



Łączniki (Joins)

Spark wykonuje joins na dwa sposoby:

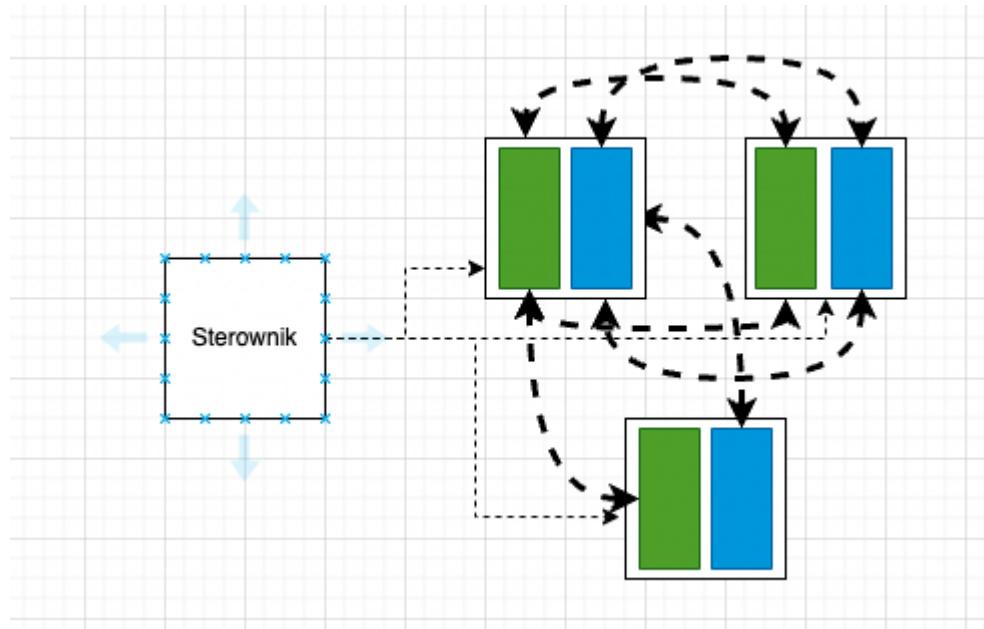
Strategia komunikacji **node-to-node communication** oraz **per node computation**

W przypadku wykonania ‘shuffle join’ wystąpi komunikacja ‘all-to-all’, czyli wymiana danych pomiędzy wszystkimi wykonawcami.

Drugi typ to ‘**BroadCast Join**’, kiedy następuje kopowanie całych tabel.

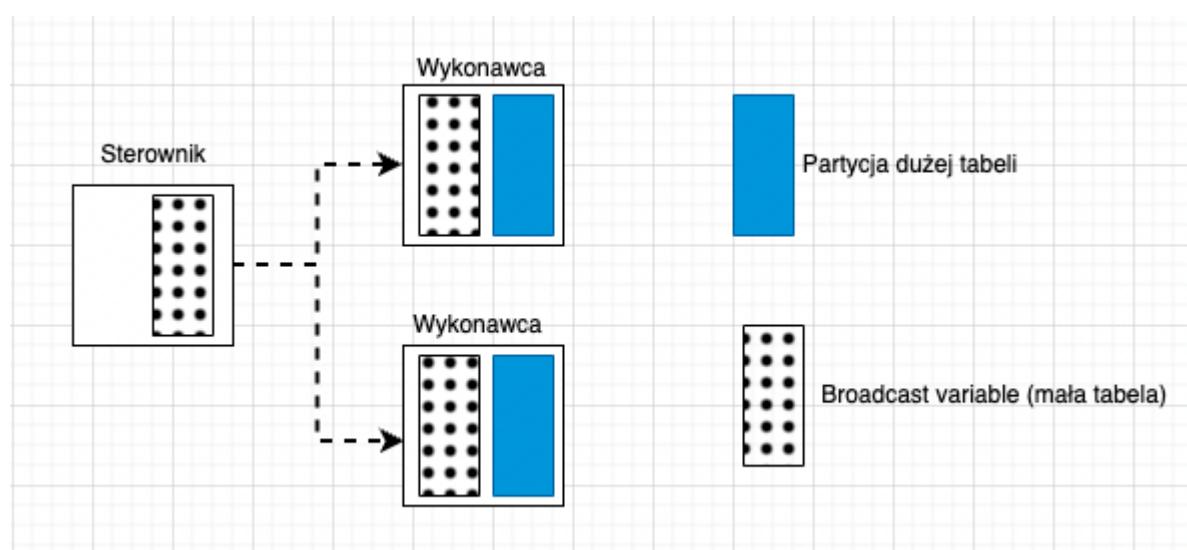
Duża tabela do dużej tabeli

Każdy węzeł komunikuje się z każdym innym węzłem i udostępnia dane zgodnie z tym, który węzeł ma określony klucz lub zestaw kluczy (używane do połączenia). Te połączenia są drogie, ponieważ mogą one przeciążyć sieć.



Duża tabela do małej tabeli

Gdy tabela jest wystarczająco mała, aby zmieścić się w pamięci pojedynczego wykonawcy, nastąpi replikacja DataFrame do każdego wykonawcy w węźle. Komunikacja wystąpi tylko raz na początku procesu.



Hints (podpowiedzi)

Interfejs SQL zawiera również możliwość podania wskazówek do wykonania połączeń. Nie są one jednak wymuszane, więc optymalizator może je zignorować. Możesz ustawić jedną z tych wskazówek, używając specjalnej składni komentarza. **MAPJOIN, BROADCAST i BROADCASTJOIN ALL.**

Optymalizacja

Jeśli poprawnie podzielisz dane przed połączeniem (**partycjonowanie**), możesz uzyskać znacznie wydajniejsze wykonanie, ponieważ nawet jeśli Spark wykona Shuffle, to dane z dwóch różnych ramek DataFrames mogą się znaleźć na tej samej maszynie.

Typy Łączników

Połączenia wewnętrzne (inner joins): zachowaj rzędy z kluczami, które wychodzą po lewej i prawej stronie

Połączenia (outer join) zewnętrzne: zachowuje wiersze istniejące w lewym lub prawym zestawie danych

Lewe połączenia zewnętrzne (left outer join): zachowaj wiersze z kluczami w lewym zestawie danych

Prawe połączenia zewnętrzne (right outer join): przechowuj wiersze z kluczami w prawym zbiorze danych

Lewe pół-złączenia (left semi join): trzymaj wiersze po lewej, a tylko po lewej stronie, gdzie klucz pojawia się w prawym zbiorze danych

Lewy łącznik anty (left anti join): trzymaj wiersze po lewej i tylko po lewej stronie, gdzie nie pojawiają się w prawym zbiorze danych

Połączenia naturalne (natural join): wykonaj połączenie, domyślnie dopasowując kolumny między dwoma zestawami danych o tych samych nazwach

Połączenia krzyżowe (cross join): dopasowuje każdy wiersz w lewym zbiorze danych z każdym rzędem w prawym zbiorze danych.

Duplikaty kolumn

W DataFrame każda kolumna ma unikatowy identyfikator w Spark. Ten unikalny identyfikator jest czysto wewnętrzny i nie jest czymś, do czego można bezpośrednio się odnieść.

Po połączeniu dwóch DataFramów, kolumna łącząca będzie zduplikowana. Na ten problem są dwa rozwiązania.

Rozwiązanie 1

Zmień wyrażenie z Boolean na string lub Seq.

`person.join(gradProgramDupe, "graduate_program").`

Rozwiązanie 2

Po wykonaniu połączenia usuń zduplikowaną kolumnę. Odnieś się do kolumny używając oryginalnej DataFrame.

Linki

- <https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-joins.html>
- <https://books.japila.pl/apache-spark-internals/broadcast-variables/?h=broadcast#resources>
- <https://databricks.com/session/optimizing-apache-spark-sql-joins>

Wspierane typy

- <https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-joins.html#join-types>
- <https://medium.com/@achilleus/https-medium-com-joins-in-apache-spark-part-2-5b038bc7455b>

Łączniki Broadcast

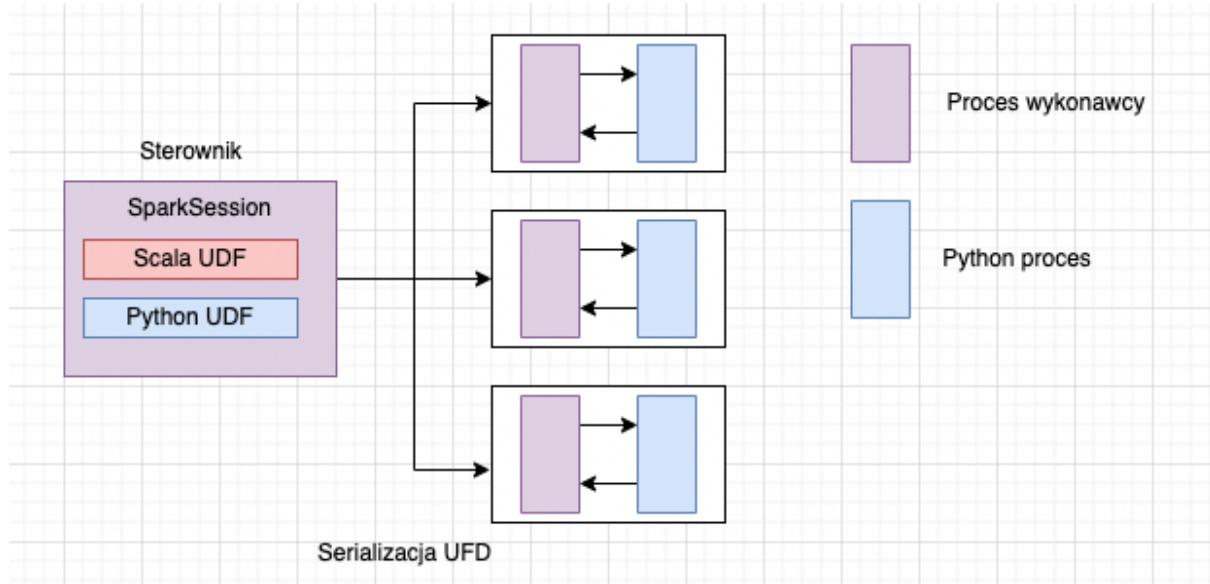
- <https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-joins-broadcast.html>

UDFs Funkcje zdefiniowane przez użytkownika

Są to dodatkowe funkcje, które możesz sam zdefiniować. Umożliwiają pisanie własnych niestandardowych przekształceń za pomocą Pythona lub Scali, a nawet korzystanie z bibliotek zewnętrznych. UDF mogą przyjmować i zwracać jedną lub więcej kolumn jako dane wejściowe. Spark UDF są niezwykle wydajne, ponieważ można je pisać w kilku różnych językach programowania.

Funkcje są rejestrowane jako funkcje tymczasowe, uruchomione w konkretnej sesji SparkSession.

Funkcje trzeba zarejestrować w Spark, żeby były dostępne na wszystkich maszynach roboczych. Spark serializuje funkcję w sterowniku i prześle ją przez sieć do wszystkich procesów executora.

**UWAGA!!**

Użycie UDF w Pythonie jest kosztowne. Najwięcej będzie kosztować serializacja danych do Pythona. W przypadku Pythona - Spark nie może zarządzać pamięcią pracownika. Oznacza to, że potencjalnie możesz spowodować awarię wykonawcy, jeśli stanie się on ograniczony zasobami (ponieważ zarówno JVM, jak i Python konkurują o pamięć na tej samej maszynie)

Żeby zdefiniować UDF wymagane są trzy korki

1. Definiujesz funkcję

Chce się pozbyć kropki z daty 12.03.2022 i zamienić je na “-”

```
def replaceDot(column: String): String ={
    column.replace(".", "-")
}
```

2. Rejestrujesz funkcję w sparku

```
val replaceDotUDF = spark.udf.register("replaceDot", replaceDot _)
```

3. Zaczynasz jej używać 😊

```
moviesDf.select(replaceDotUDF(col("date_published"))).show
```

| replaceDot(date_published) |
|----------------------------|
| 01-01-1987 |
| 06-05-1991 |
| 01-05-1987 |
| 01-03-1987 |
| 18-03-1987 |

Linki

- <https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-udfs.html>

Funkcje

Window - umożliwia określenie jednego lub więcej kluczy, a także jednej lub więcej funkcji agregacji w celu przekształcenia kolumn wartości. Jednak wiersze wejściowe do funkcji są w jakiś sposób powiązane z bieżącym wierszem.

Grouping Set, którego można użyć do agregowania na wielu różnych poziomach. Zestawy grupujące są dostępne jako element podstawowy w SQL oraz za pośrednictwem zestawień i kostek w DataFrames.

Rollup umożliwia określenie jednego lub więcej kluczy oraz jednej lub więcej funkcji agregacji w celu przekształcenia kolumn wartości, które zostaną podsumowane hierarchicznie.

Cube pozwala określić jeden lub więcej kluczy oraz jedną lub więcej funkcji agregacji w celu przekształcenia kolumn wartości, które zostaną podsumowane we wszystkich kombinacjach kolumn.

Funkcje okienkowe

Funkcja ‘groupBy’ pobiera dane, a każdy wiersz może należeć tylko do jednej grupy. Funkcja okna oblicza wartość zwracaną dla każdego wejściowego wiersza tabeli na podstawie grupy wierszy zwanej ramką. Każdy wiersz może należeć do jednej lub kilku ramek.

Spark obsługuje trzy rodzaje funkcji okna: **funkcje rankingowe, funkcje analityczne i funkcje agregujące**.

| | SQL | DataFrame API |
|---------------------------|--------------|---------------|
| Ranking functions | rank | rank |
| | dense_rank | denseRank |
| | percent_rank | percentRank |
| | ntile | ntile |
| | row_number | rowNumber |
| Analytic functions | cume_dist | cumeDist |
| | first_value | firstValue |
| | last_value | lastValue |
| | lag | lag |
| | lead | lead |

Linki

- <https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html>
- <https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-functions-windows.html>

Przesyłanie strumieniowe

Porównanie

| Spark Structured Streaming | Spark Streaming |
|---|---|
| <ul style="list-style-type: none"> • Zbudowany na silniku Spark SQL • Oparty na Tabeli • Dataset/DataFrame API | <ul style="list-style-type: none"> • Rozszerzenie do Spark API • Oparty na Micro-batch • DStream (stream bloków RDD) |

Linki

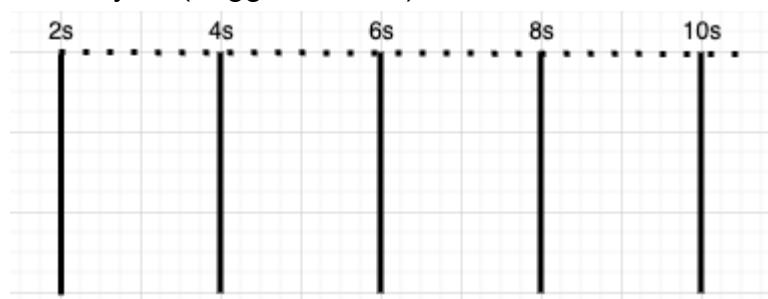
- <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>
- [Structured Streaming - Azure Databricks | Microsoft Docs](#)
- <https://spark.apache.org/docs/latest/streaming-programming-guide.html>

Strumienie ustrukturyzowane

Problem ze Spark Streaming - Micro-batch

- Dane napływają z TCP-IP np Kafka, Kinesis, inne źródła
- Dane wpływają szybciej niż jesteś w stanie je przetworzyć

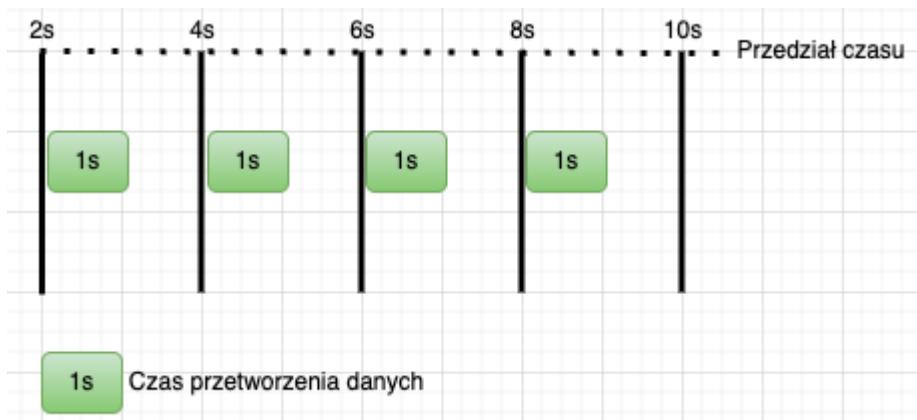
Rozwiązywanie problemu model Micro-Batch czyli zbieramy dane w przedziałach czasowych (Trigger Interval)



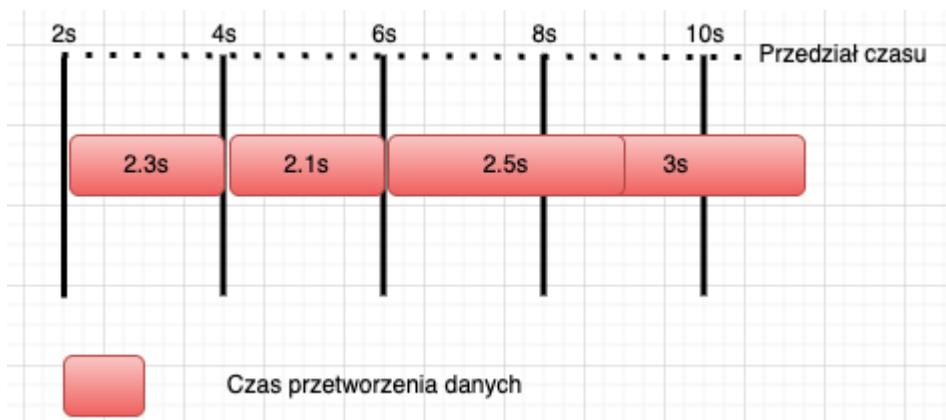
Przetwarzanie micro-batcha

Dla każdego przedziału czasowego przetwarzamy dane z poprzedniego przedziału czasowego (np ostatnie dwie sekundy)

W tym przykładzie przetwarzamy dane z dwóch sekund w jedna sekunde



Opóźnienie

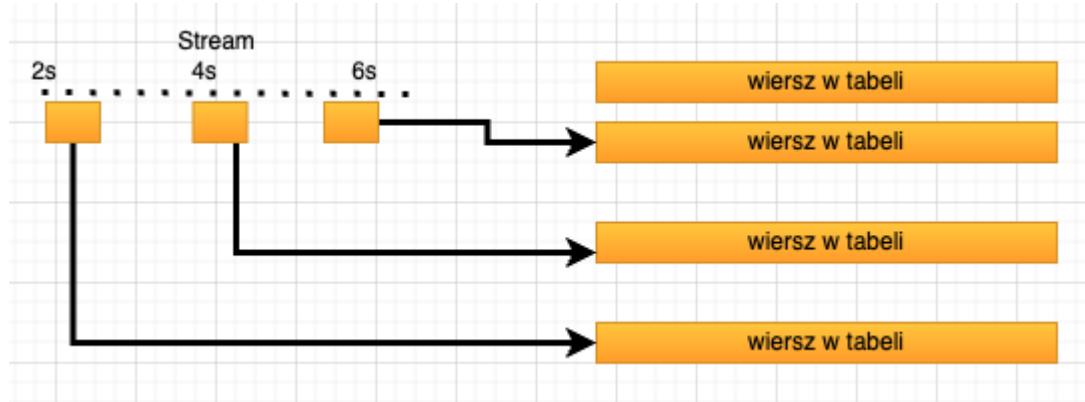


Mikro-batch alternatywa

CEL: przetworzenie danych z poprzedniego przedziału zanim wpłyną kolejne dane

Rozwiążanie: Spark Structured Streaming traktuje **stream** (micro batch) jako **tabele** i ciągle dodaje (**append**) nowe dane.

Trigger interval = append (new row)



Zapytania do tabeli

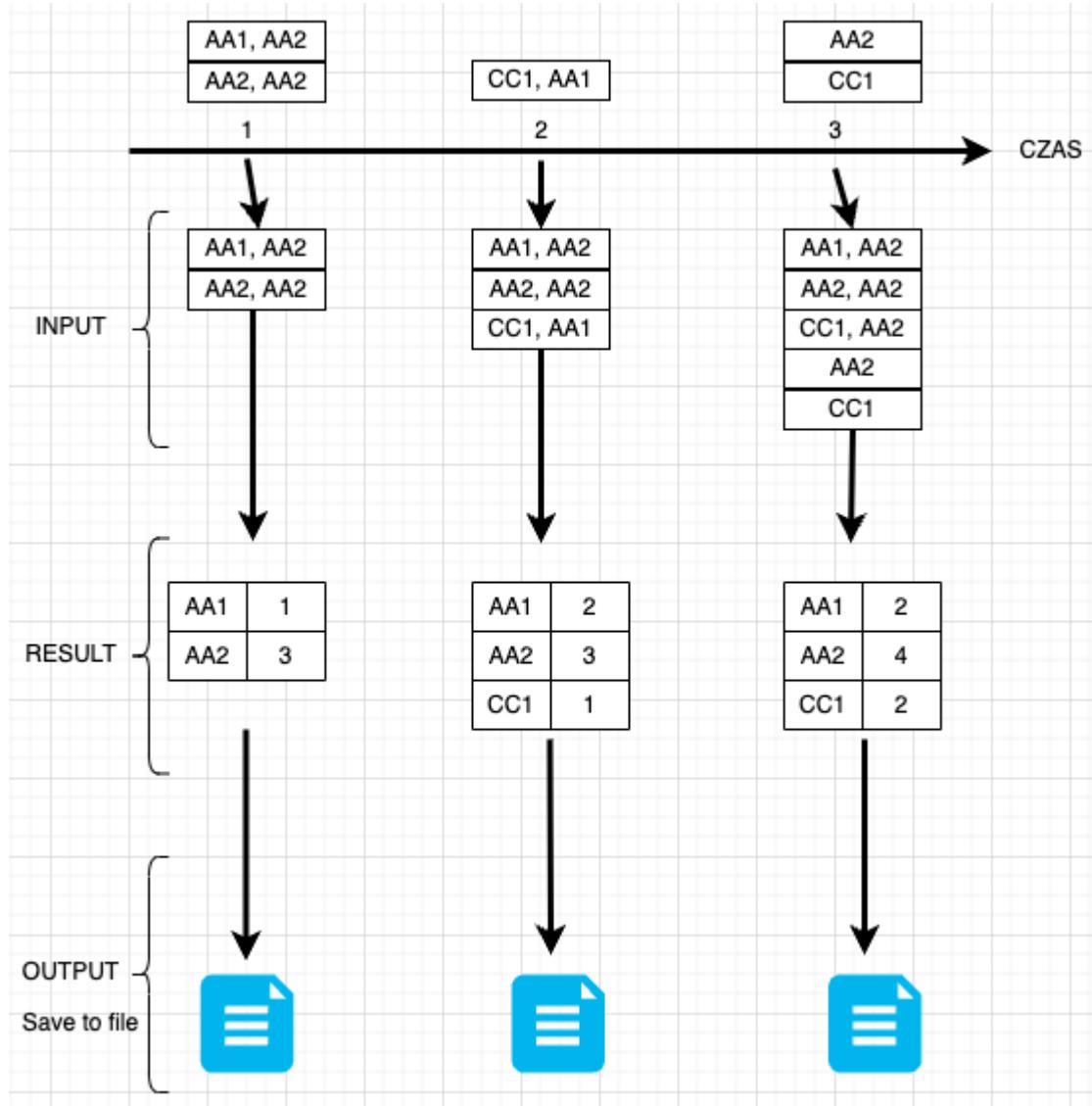
Proces pobierania danych z tabeli:

Dla każdego przedziału czasu dane są dodawane do tabeli

1. **Input table** - definiujemy zapytanie jak na tabelę statyczną
2. **Result table** - wyliczenia są przekazywane do tabeli wyników
3. **Sink** - Na koniec zapis do output

Definiujesz trigger kiedy input table powinna być uaktualniona.

Przy każdym wywołaniu triggera spark sprawdza nowe dane i inkrementalnie uaktualnia result table.



Tryby wyjściowe (Output Modes)

Append: **Tylko nowe wiersze** dołączone do tabeli wyników od ostatniego wyzwalacza są zapisywane do sinka. Jest to przydatne tylko wtedy, gdy istniejące wiersze w tabeli wyników nie zostały zmienione.

Complete: Cała zaktualizowana tabela wyników jest zapisywana w sinku.

Update: Tylko wiersze w tabeli wyników, które zostały zaktualizowane od ostatniego wyzwalacza, zostaną wyprowadzone do sinka.

Stream części składowe

Źródła:

File source

- Kafka
- Azure Event hub / AWS Kinesis
- Pliki w systemie HDFS
- TCP-IP sockets

Sink

- Kafka
- Azure Event Hub/AWS Kinesis
- File
- Console
- Foreach
- ForeachBatch

Output Sinks

| Typ | Output Mode | Opis |
|---------------|--------------------------|---|
| File | Append | Format dostępne w DataFrameWriter |
| Kafka | Append, Complete, Update | Output dodany do topic Kafka |
| Console | Append, Complete, Update | Output do konsoli, debugowanie |
| Memory | Append, Complete | In-memory table (Spark SQL lub DataFrame) |
| Foreach | Append, Complete, Update | Własna definicja sinka dla każdego wiersza |
| Foreach Batch | Append, Complete, Update | Dowolne operacje i niestandardowa logika na wyjściu |

Charakterystyka Streamu

- Stream nie są posortowane
- Dane można przesłać do wielu sinków data lake
- Dane wpływają szybciej niż jesteś je w stanie przetworzyć
- Dane zawierają 'event-time' - czas wytworzenia danych
- Stream nie musi być ciągły

- Np. pliki logów mogą być przetwarzane co godzinę
- Dane finansowe mogą być przetwarzane raz na miesiąc

Odporność na błędy (Fault tolerance)

- W każdej chwili proces może być zrestartowany i dane przetworzone ponownie.
- Zakłada się, że każde źródło strumieniowe ma przesunięcia (offset) - śledzi odczytaną pozycję w strumieniu.
- Używa mechanizmu 'checkpoint'
- Zapisuje zakres przetwarzanych danych w każdym wyzwalaczu (trigger)
- Odtwarzalne źródła i 'idempotent sink'

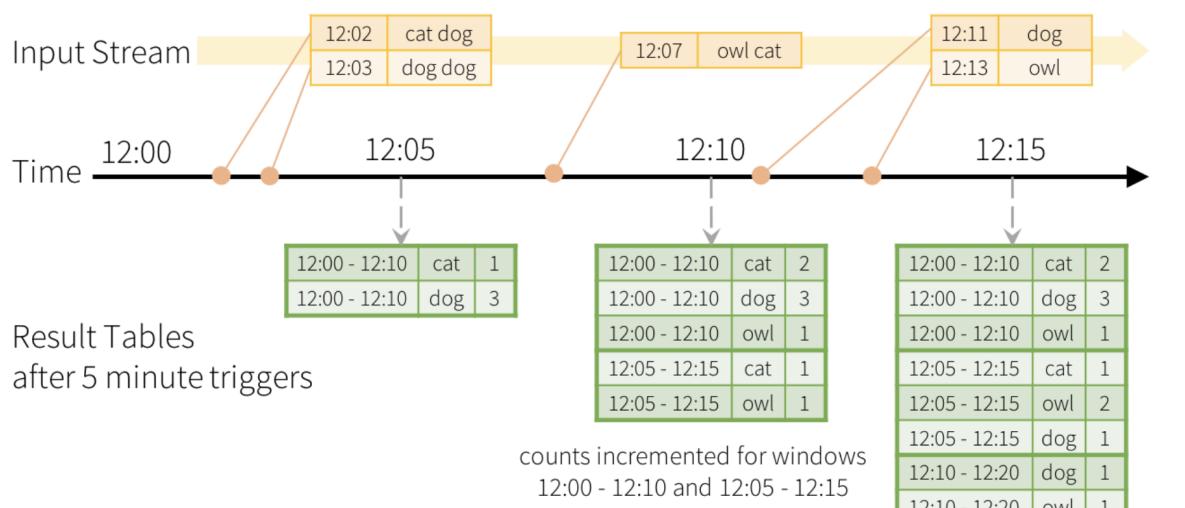
Schemat

Musisz podać schemat kiedy streamujesz z pliku

Dla potrzeb testowania ad-hoc możesz włączyć wczytywanie schematu **spark.sql.streaming.schemaInference true**.

Windowing

Jest to agregacja wartości na podstawie czasu zdarzenia (event-time) w grupie.

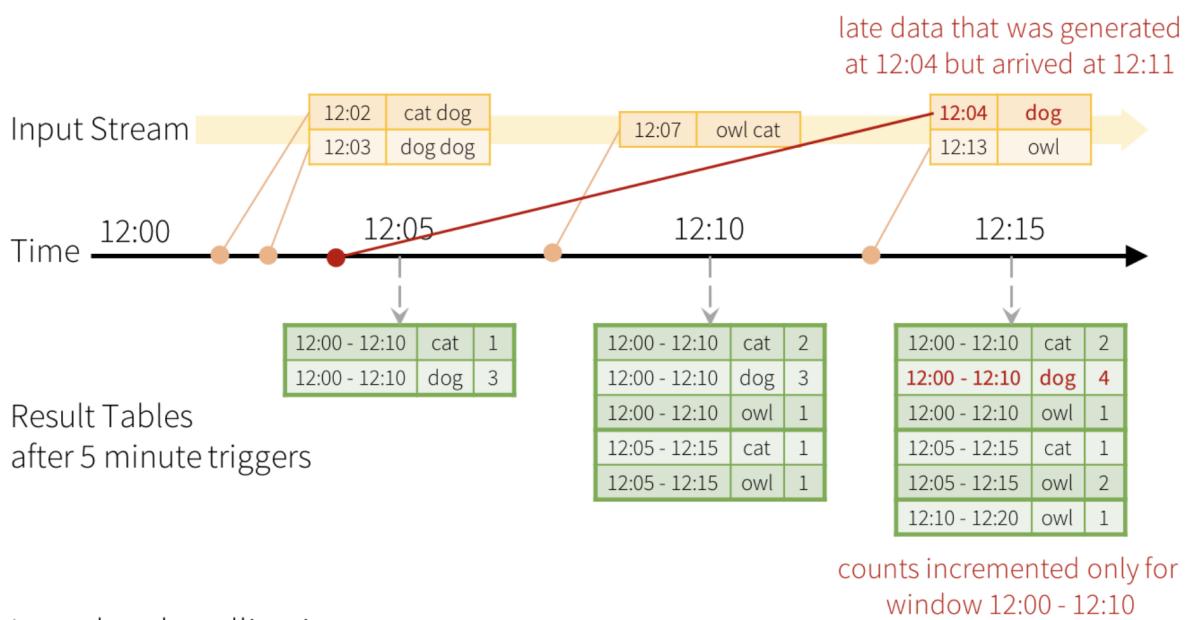


Windowed Grouped Aggregation
with 10 min windows, sliding every 5 mins

Spóźnione wydarzenia

Watermark to pośredni stan w pamięci kiedy steam przestaje trzymać dane tymczasowe.

Dane zostaną uaktualnione w bloku danych zgodnie z czasem wygenerowania.



Watermark

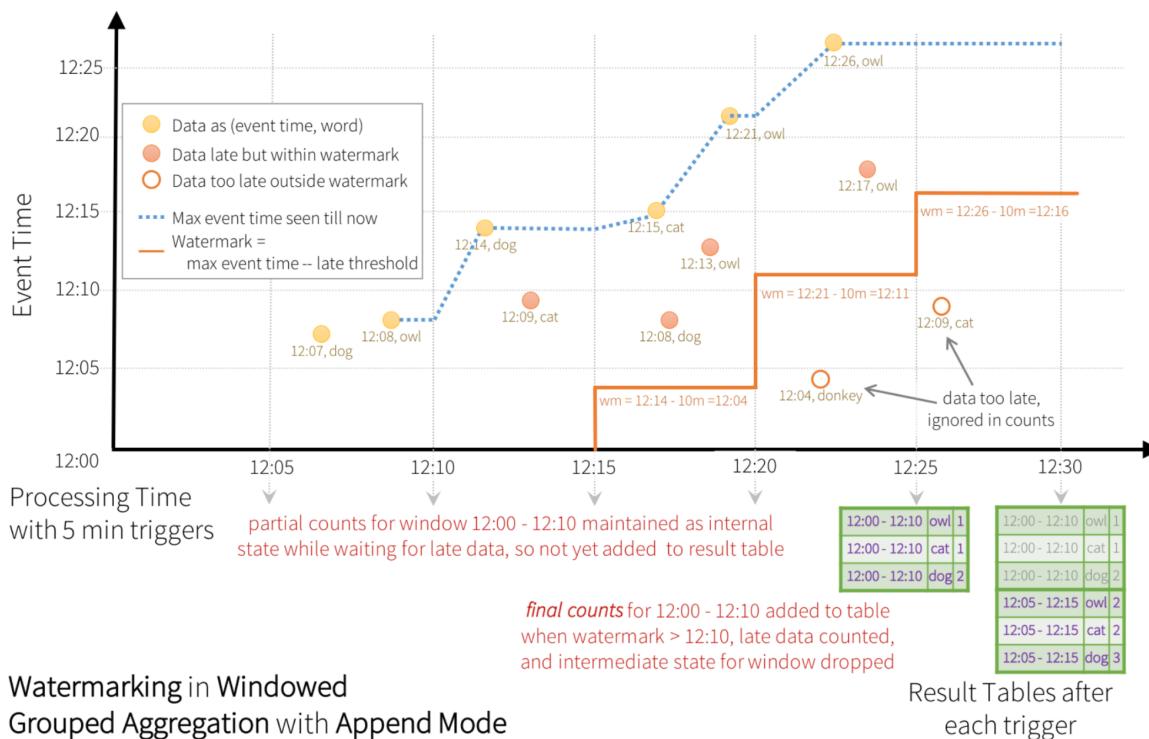
Automatycznie śledzi aktualny czas zdarzenia w danych i stara się odpowiednio wyczyścić stary stan.

Mozesz zdefiniować watermark zapytania, określając kolumnę czasu zdarzenia i próg oczekiwanej opóźnienia danych pod względem czasu zdarzenia.

(max event time seen by the engine - late threshold > window ending at time T)

Akceptuje dane spóźnione 10 minut

```
val windowedCounts = words
  .withWatermark("timestamp", "10 minutes")
  .groupBy(
    window($"timestamp", "10 minutes", "5 minutes"),
    $"word")
  .count()
```



Warunki dla watermark

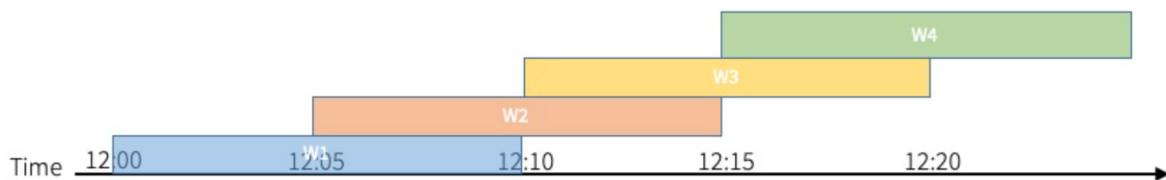
- Output mode musi być **Append** or **Update**.
- Nie możesz użyć **Complete** mode on potrzebuje zachować wszystkie agregacje - szczegóły w dokumentacji - [Output Modes](#)
- Agregacje muszą mieć kolumnę '**event-time**' lub ustawione okienko (window) na kolumnie 'event-time'.
- `withWatermark` musi być użyte na tej samej kolumnie (timestamp) użytej do agregacji. **Przykład błędu**, `df.withWatermark("time", "1 min").groupBy("time2").count()`.
- `withWatermark` musi być ustawiony przed agregacją. **Przykład błędu**, `df.groupBy("time").count().withWatermark("time", "1 min")`.

Typy okienek Spark

Tumbling Windows (5 mins)



Sliding Windows (10 mins, slide 5 mins)



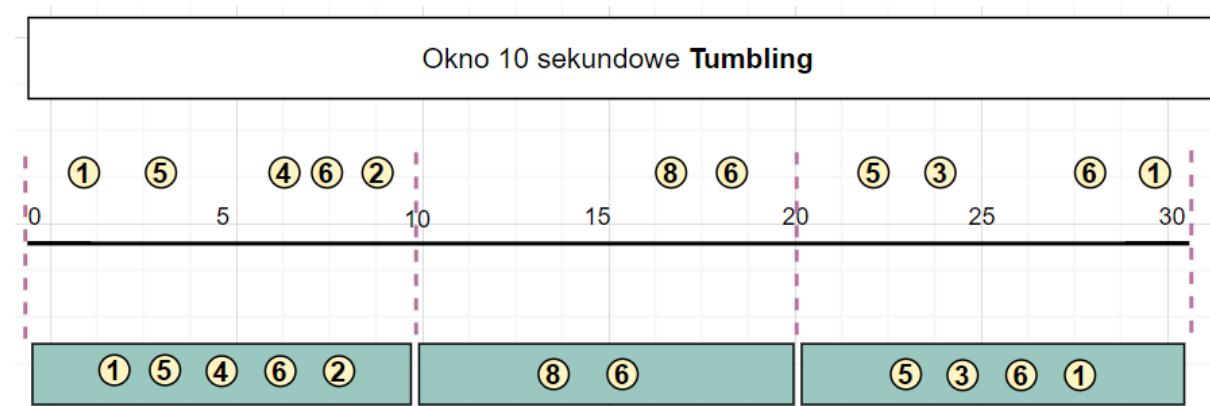
Session Windows (gap duration 5 mins)



Tumbling Window

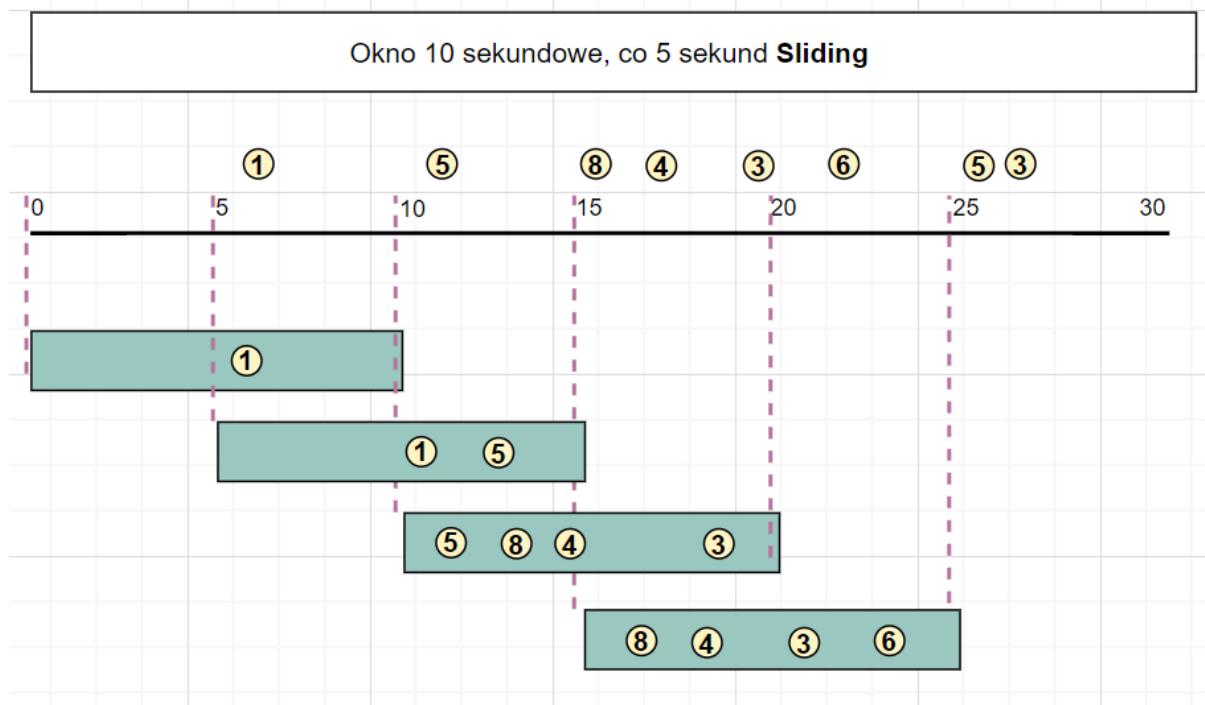
Jest to seria, nie nakładających się i ciągłych przedziałów czasowych o stałych rozmiarach. Dane wejściowe wchodzą tylko do jednego okna.

Zgodnie z powyższym przykładem co 5 minut policz wszystkie wydarzenia tylko dla tego przedziału czasowego.



Sliding Window

Jak w przypadku okna Tumbling jest ono stałych rozmiarów ale mogą na siebie zachodzić, okno ma stałą długość i uruchamia się regularnie.



Session Window

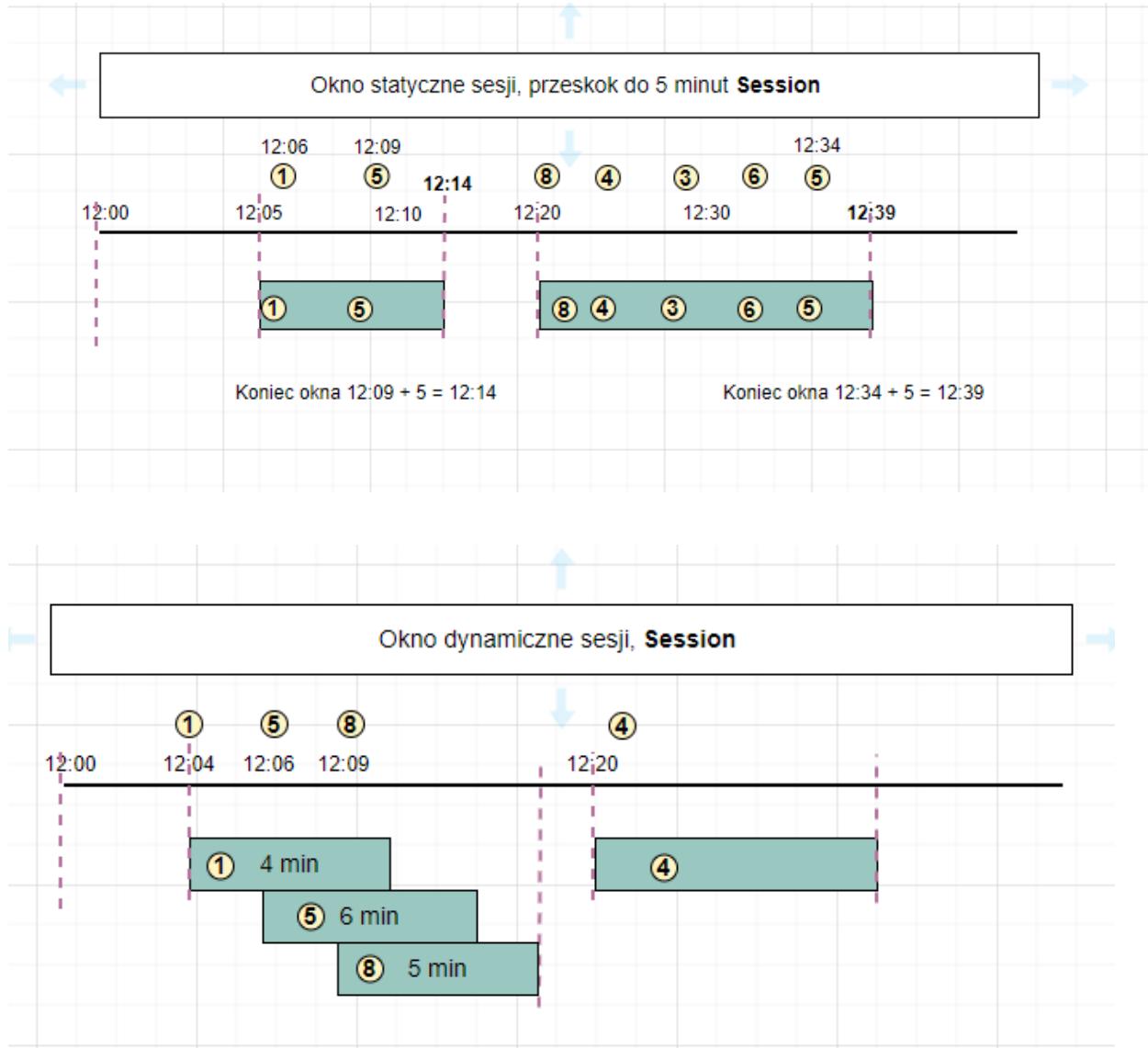
Są dwa typy okna sesji: **statyczny** i **dynamiczny**.

Okno sesji jest dynamiczne i zmienia się, w zależności od danych wejściowych. Rozpoczyna się od wydarzenia i kontynuuje, jeśli kolejne dane wejściowe zostały odebrane w czasie trwania przerwy.

W przypadku statycznego czasu trwania przerwy okno sesji zamyka się, gdy nie ma żadnych danych wejściowych otrzymanych od poprzedniego wydarzenia.

Najlepszy przykład to logowanie się do banku, jeśli nie będziesz aktywny przez jakiś czas zostaniesz wylogowany, chyba że jesteś aktywny.

W oknie dynamicznym każde wydarzenie jest traktowane jako osobna sesja. Wszystkie okna w sesjach, które nakładają się czasowo są łączone w jedną całość.



Linki:

- [Structured Streaming Programming Guide - Spark 3.3.0 Documentation \(apache.org\)](https://spark.apache.org/docs/3.3.0/structured-streaming-programming-guide.html#session-windows)
- <https://www.databricks.com/blog/2021/10/12/native-support-of-session-windows-in-spark-structured-streaming.html>

Usuwanie duplikatów

Możesz usuwać duplikaty strumieniach danych, używając w zdarzeniach unikalnego identyfikatora.

W celu usunięcia duplikatów, możesz użyć watermark lub usunąć duplikaty przy użyciu guid.

```
// Without watermark using guid column
streamingDf.dropDuplicates("guid")

// With watermark using guid and eventTime columns
streamingDf
  .withWatermark("eventTime", "10 seconds")
  .dropDuplicates("guid", "eventTime")
```

Linki:

<https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#streaming-deduplication>

Zabronione operacje

- Wiele agregacji strumieniowych (ex łańcuch agregacji).
- Nie działa `Limit()` and `take(n wierszy)`.
- Operacje `Distinct()`.
- Usuwanie duplikatów wykonanych po agregacji.
- Sortowanie tylko po agregacji i w `Complete Output Mode`.
- Kilka typów outer joins. [Pełna lista operacji Join](#).

Odzyskiwanie po awarii - Checkpointing

Ten mechanizm zapewnia, odtworzenie procesu jeśli wydarzy się jakaś awaria. Zasada działania jest dość prosta. Musisz podać lokalizację, gdzie Spark będzie zapisywać aktualny stan procesu. Są to informacje dotyczące zakresu danych,产生的 w każdym wyzwalaczu.

- Checkpointing - czyli logi z punktami kontrolnymi
- W przypadku awarii lub wyłączenia streamu można go uruchomić ponownie.
- Sink - idempotencja wiele zapisów tych samych danych nie spowoduje duplikatów
- Lokalizacja musi być zgodna ze standardem HDFS
- Zapisuje产生的 aggregacje w każdym wyzwalaczu (trigger)
- Kiedy następuje awaria silnik stara się uruchomić proces ponownie
- Działa tylko na źródłach ([replayable](#)) np Kafka czy File

Przykłady Spark Streaming

Streaming i manipulacja DataFrame. Poniżej przykłady kodu użycia, window, checkpoint, agregacji.

Tworzenie streamu.

Stream możesz stworzyć z wielu źródeł, takich jak pliki, json, csv. Ze źródeł takich jak Kafka. Poniżej przykład 'socket' możesz go użyć tylko do testowania ponieważ nie zapewnia mechanizmu odporności na awarie.

```
val streamDF = spark.readStream
  .format("socket")
  .option("host", "localhost")
  .option("port", 9999)
  .load()
```

Liczenie

```
from pyspark.sql.functions import *
countsDF = streamDF.agg(count("*"))
```

Pamiętaj, że to DataFrame to cały czas DataFrame a nie to stream czy nie to dalej możesz wykonywać te same operacje co na zwykłej ramce danych. Funkcje ze zwykłej ramki danych będzie działać na strumieniu np. select, czy filter.

Dodaję kolumnę i formatuję czas

```
cleanDF = (streamDF
  .withColumn("time_string", col("value").substr(2, 23))
  .withColumn("epoch", unix_timestamp("ts_string", "yyyy/MM/dd HH:mm:ss.SSS"))
  .withColumn("capturedAt", col("epoch").cast("timestamp"))
)
```

Funkcja Window agreguje dane w wybranym przedziale czasowym, odpowie na pytanie ile wydarzeń (events) przychodzi co 10 sekund.

```
secondDF = (cleanDF
  .select("capturedAt")
  .groupBy( window(col("capturedAt"), '10 second'))
  .count()
  .orderBy("window.start")
)
```

Użycie watermark. W powyższym przykładzie powstaje dużo okienek (windows). Każde okienko musi być przetrzymywane i zarządzanie co wymaga dużych zasobów. Z czasem wszystko zacznie spowalniać. Watermark zapewni, że “stare” okienka zostaną usunięte.

```
windowDF = (filterDF
    .withWatermark("capturedAt", "1 minute")
    .groupBy(window(col("capturedAt"), "1 second"))
    .count()
    .select( col("window.start").alias("start"),
        col("window.end").alias("end"))
    .orderBy(col("start"))
)
```

Checkpointing - w razie przerwania strumienia lub awarii jest możliwość odtworzenia stanu i powrót do normalnej funkcjonalności.

```
streamingQuery = (windowDF
    .writeStream
    .outputMode("complete") # Zapis danych
    .option("checkpointLocation", path) # Checkpoint ścieżka
    .format("memory") # Zapis w pamięci
    .queryName("zapytanie") # Nazwa zapytania
    .trigger(processingTime="20 seconds") # Trigger co 20 sekund
    .start() # Uruchamia zapis
)
```

Monitorowanie

Co można monitorować.

Spark Aplikacje i Jobs:

Spark UI i Logi

JVM:

Indywidualne VMki,
jstack (stack trace), jmap (heap-dumps),
jstat (statystyki), jconsole (wizualizacja), jvisualvm (profilowanie spark job)

Host OS:

CPU, memory, network, I/O (dstat, iostat, iotop)

Klaster:

YARN, Mesos, Standalone. Narzędzia **Ganglia, Prometheus**

Monitoruj Driver

Spark daje ci dostęp do wielu informacji w postaci UI, gdzie znajdziesz najważniejsze informacje dotyczące aplikacji i tego co dzieje się wewnątrz sparka.

Spark ma konfigurowalny system oparty na bibliotece [Dropwizard](#). Lokalizacja konfiguracji: **\$SPARK_HOME/conf/metrics.properties**. Można zmienić lokalizację tego pliku przy użyciu: spark.metrics.conf

Output można przesyłać do Ganglia

Logi Sparka

Dodaj logi do aplikacji

<https://github.com/databricks/Spark-The-Definitive-Guide/tree/master/project-templates/scala>

Przykład pozwoli na dodanie logów aplikacji do logów Spark

Zmiana poziomu logowania w Spark

spark.sparkContext.setLogLevel("INFO")

Spark UI > Driver Logs > Log4j Output

Poniżej przykład

Log4j output

```
22/06/05 09:55:12 INFO MetastoreMonitor: Metastore healthcheck success
22/06/05 09:55:23 INFO DriverCorral: DBFS health check ok
22/06/05 09:55:23 INFO HiveMetaStore: 0: get_database: default
22/06/05 09:55:23 INFO FileInputFormat: Input paths for partition: part-00000
```

Spark UI

SparkContext dostarcza web UI na porcie **4040**,

Lokalny przykład **http://localhost:4040**

Dla wielu aplikacji Spark zwiększy numer portu (4021, 4022...)

Statystyki Stage

DAG Directed Acyclic Graph każde niebieskie pole reprezentuje stage, wszystkie składają się na Job

Poniżej proces, który jest pokazany na zrzucie ekranu Screen 1 i Screen 2

1. Skan pliku, podaje ilość wierszy

2. Exchange - wykonałem repartition
3. WholeStateCodeGen pełny skan pliku
4. Filtrowanie 6 wierszy i agregacja dla każdej partycji
5. Końcowy etap z 3 wierszami

localhost:4040/SQL/execution/?id=1

Apache Spark 3.1.2

Jobs Stages Storage Environment Executors **SQL**

Details for Query 1

Submitted Time: 2022/06/05 12:29:53
Duration: 5 s
Succeeded Jobs: 1

Show the Stage ID and Task ID that corresponds to the max metric

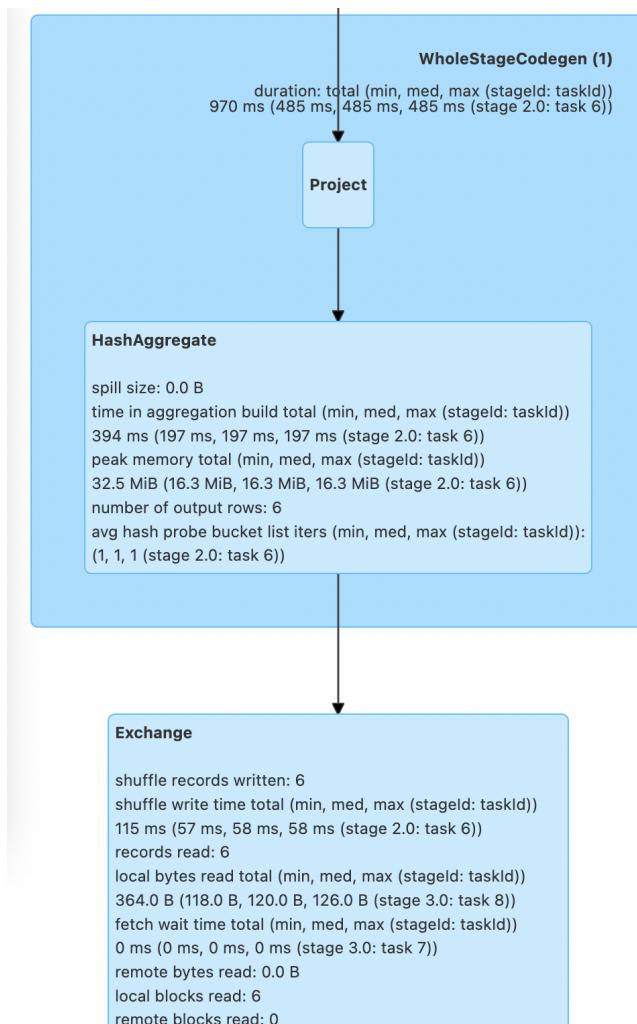
Scan csv

number of output rows: 541,909
 number of files read: 1
 metadata time: 2 ms
 size of files read: 43.0 MiB

Exchange

shuffle records written: 541,909
 shuffle write time total (min, med, max (stageId: taskId))
 65 ms (7 ms, 8 ms, 43 ms (stage 1.0: task 4))
 records read: 541,909
 local bytes read total (min, med, max (stageId: taskId))
 1034.7 KiB (516.6 KiB, 518.1 KiB, 518.1 KiB (stage 2.0: task 6))
 fetch wait time total (min, med, max (stageId: taskId))
 0 ms (0 ms, 0 ms, 0 ms (stage 2.0: task 6))
 remote bytes read: 0.0 B
 local blocks read: 8
 remote blocks read: 0
 data size total (min, med, max (stageId: taskId))
 23.8 MiB (4.3 MiB, 6.5 MiB, 6.5 MiB (stage 1.0: task 1))
 remote bytes read to disk: 0.0 B
 shuffle bytes written total (min, med, max (stageId: taskId))
 1034.7 KiB (227.7 KiB, 267.9 KiB, 273.1 KiB (stage 1.0: task 2))

Screen 1



Screen 2

Jobs

Details for Job 1

Status: SUCCEEDED
Submitted: 2022/06/05 12:39:22
Duration: 4 s
Associated SQL Query: 1
Completed Stages: 3

| Submitted | Duration | Tasks: Succeeded/Total |
|---------------------|----------|------------------------|
| 2022/06/05 12:39:25 | 2 s | 200/200 |
| 2022/06/05 12:39:24 | 0.6 s | 2/2 |
| 2022/06/05 12:39:22 | 2 s | 4/4 |

1. Pierwszy stage został podzielony na 4 taski (cory)
2. Wykonanie Repartition wywołało 2 zadania (tasks)
3. Ostatni to 200 bo tyle jest partycji shuffle

Linki:

- [Spark UI Konfiguracja](#)
- [REST API http://localhost:4040/api/v1](http://localhost:4040/api/v1)

Wolne Taski

Bardzo częstym problemem są wolne zadania. Może to być spowodowane nierównomiernym rozłożeniem danych w poszczególnych partycjach, tzn część wykonawców ma więcej danych niż inne.

Metody na rozwiązywanie tego problemu

- Zwięksź ilość partycji,
- Wykonaj Repartition na innych kolumnach,
- Zwięksź ilość pamięci dostępnej dla wykonawcy
(spark.conf.get("spark.executor.memory"))
- Sprawdź joiny i agregacje
- Sprawdź UDFs
- Dataset (tworzenie obiektów) - sprawdź GC metryki

Wolne agregacje

Kiedy zaobserwujesz wolno działające agregacje to sprawdź poniższą czeklistę. Powód może być znacznie więcej, i nie da się wyjaśnić wszystkich ale może trafisz na ten bardziej podstawowy.

- Wszystkie filtry i SELECT powinny być przed agregacją
- Uwaga na groupBy, powoduje shuffle
- Zwiększenie ilości partycji przed agregacją może zredukować ilość kluczy w taskach
- Brak wystarczającej pamięci spowoduje 'spill to disk'
- Spark optymalizuje i przeskakuje wartości null
- Nie pobieraj do sterownika (driver) collect, collect_list, collect_set

Wolne joiny

- Wybierz odpowiedni join (shuffle join, broadcast join), poszczególne typy łączników mogą mieć różny wpływ na osiągi.
- Partycjonowanie danych przed wykonaniem join, szczególnie jeśli dane będą łączone wiele razy.
- Filtry i Select powinny być wykonane przed joinem, celem redukcji ilości danych.
- Użyj wartości null a nie pustego stringa lub słów " " "EMPTY" "null"
- Użyj broadcast join tam gdzie to ma sens

Wolne Odczyty i Zapisy

Te problemy występują rzadziej od powyższych ale warto je mieć na uwadze.

- Włącz ‘speculation’ **spark.speculation true** (uruchamia dodatkowe taski żeby sprawdzić błędy ‘transient’, uwaga na duplikaty przy zapisie)
- Sprawdź przepustowość sieci (bandwidth)
- Dla storage HDFS sprawdź czy spark widzi ten sam hostname dla nodów co system plików (kolumna ‘locality’ in Spark UI)

Błędy OutOfMemory

Te błędy występują znacznie częściej, poniżej kilka przykładów które mogą spowodować błędy OOM

- Driver lub Executor jest przeładowany i nie ma wolnej pamięci
- Zbierasz dużo danych do drivera
- Broadcast join na dużej tabeli (broadcast threshold jest konfigurowalny)
- Długo działająca aplikacja może stworzyć wiele obiektów. **Jmap** pokaże obiekty, monitoruj JVM
- Zwiększa ilość pamięci, konfigurowalne dla sterownika i wykonawcy
- Użycie innego języka ‘language binding’ Python konwersja danych
- Uwaga na dzielenie się SparkContext z innymi
- Uwaga na UDFs

Książki

[Spark: The Definitive Guide: Big Data Processing Made Simple](#)

[Learning Spark: Lightning-Fast Big Data Analysis](#)

Ciekawe Blogi

[The Internals of Apache Spark](#)

[seequality](#)

[jameserra.com](#)

[mrpaulandrew.com](#)

[marczak.io](#)

[cloudarchitected.com](#)

[Databricks engineering blog](#)

[Microsoft Azure Blog](#)

[Microsoft Learn](#)

[Mikulski Bartosz](#)

[Big Data and Analytics](#)

[Big Data Quarterly](#)

[Database Weekly by SQLServerCentral.com](#)

[Power BI Weekly Newsletter](#)

[Azure Weekly Newsletter](#)

[Brent Ozar SQL Server](#)

[SQLBI Articles](#)

[Data Mesh Learning Newsletter](#)

[Dataversity white papers](#)

Lakehouse

Co to jest Lakehouse

Z terminów Data Lake + Data Warehouse powstał nowy termin Lakehouse, który ma łączyć zalety obu rozwiązań. Nie jest to nowy termin, istnieje on już od jakiegoś czasu, chodź obecnie zyskuje na popularności. Ten rodzaj architektury jest już w użyciu takich firm jak Databricks czy Snowflake.

Ideą Data Lakehouse jest połączenie zalet hurtowni danych oraz data lake. A więc mamy struktury danych i model zarządzania tak jak przy hurtowni. Lakehouse jest bardziej atrakcyjny dla analityków (Data Scientist), ponieważ daje więcej możliwości i elastyczności w analizie danych. Ta elastyczność ułatwia wykorzystanie nauczania maszynowego. Drugim plusem jest koszt, takie dane trzymane w luźnej formie kosztują znacznie mniej niż relacyjna baza danych.

Zalety Data Lakehouse

- Równoległe przetwarzanie danych przy użyciu technologii Big Data
- Wsparcie dla schematów danych i mechanizmów zarządzania
- Bezpośredni dostęp do wszystkich danych
- Izolacja danych od warstwy obliczeniowej
- Wsparcie dla wielu formatów danych: json, parquet, avro, csv, binary
- Wsparcie dla danych IoT
- Możliwość przetwarzania strumieniowego
- Migracja pomiędzy technologiami
- Luźne dane są łatwiejsze w zarządzaniu retencją

Linki

https://databricks.com/blog/2020/01/30/what-is-a-data-lakehouse.html?utm_source=newsletter&utm_medium=email&utm_campaign=co_przyniesie_rok_2022&utm_term=2022-03-07

Lakehouse z Delta Lake

https://databricks.com/p/webinar/deep-dive-into-lakehouse-with-delta-lake-complimentary-training?utm_source=databricks&utm_medium=email&utm_campaign=7013f00000Ljo2AAC&utm_content=v1p

Dobre praktyki Databricks

<https://github.com/Azure/AzureDatabricksBestPractices/blob/master/toc.md>

<https://learn.microsoft.com/en-us/azure/databricks/>

Przykład

Do not Store any Production Data in Default DBFS Folders

Impact: High

This recommendation is driven by security and data availability concerns. Every Workspace comes with a default DBFS, primarily designed to store libraries and other system-level configuration artifacts such as Init scripts. You should not store any production data in it, because:

1. The lifecycle of default DBFS is tied to the Workspace. Deleting the workspace will also delete the default DBFS and permanently remove its contents.
2. One can't restrict access to this default folder and its contents.

| This recommendation doesn't apply to Blob or ADLS folders explicitly mounted as DBFS by the end user