

Semana 10

Objetivo: Apresentação do módulo de Apache Spark Streaming e criação de processo para processamento streaming da XPTO.

Conteúdo:

- Spark Streaming
- Spark Streaming Estruturado
- Desafio XPTO: Twitter
- Desafio XPTO: Ingestão do Twitter E Processamento Streaming

Desafio: Realizar a leitura do material completo, execute os exemplos do material e os 7 exercícios.

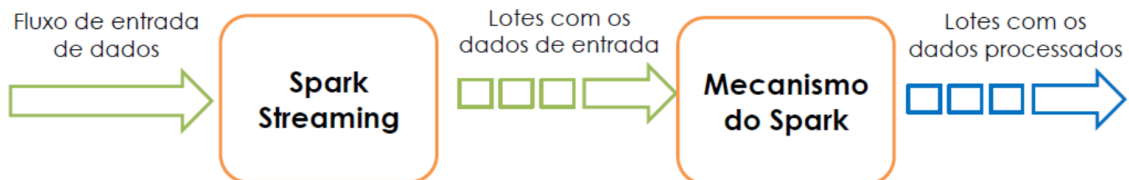
Spark Streaming

Com o avanço e expansão das tecnologias de Big Data muitas empresas mudaram o foco para a tomada de decisão com base nos dados. Assim, **quanto mais rápido as análises forem realizadas melhor**. Essa análise é realizada através da leitura dos fluxos de dados de entrada e do processamento para obter resultados que podem gerar algum **valor para o negócio**.

Spark Streaming é uma biblioteca que permite processar fluxos de dados próximo ao tempo real.

- Spark Streaming foi introduzido no Spark versão 0.7 no começo de 2013.

Tem por objetivo fornecer **uma arquitetura escalável e tolerante a falhas** que utiliza o paradigma de RDD em lote. Basicamente o Spark Streaming opera em **micro lotes (pequenos batches)** ou intervalos de lote (de pelo menos 500ms de intervalo janela maiores).



Como pode ser visualizado no diagrama anterior, Spark Streaming recebe **um fluxo de dados de entrada e internamente divide os dados em múltiplos lotes menores** (o tamanho é definido com base no intervalo de tempo). O mecanismo do Spark **processa esses lotes de dados de entrada** para um conjunto de resultados dos dados processados. O Spark Streaming possibilita o uso de **diversas fontes**.

O Spark Streaming é uma extensão da API principal do Spark que permite o processamento escalável, de alta taxa de transferência e tolerante a falhas de fluxos de dados ativos. Os dados podem ser oriundos de várias fontes, como Kafka, Flume, Kinesis ou sockets TCP, e podem ser processados usando algoritmos complexos, expressos com funções de alto nível, como *map*, *reduce*, *join* e *window*. Por fim, os dados processados podem ser enviados para sistemas de arquivos, bancos de dados e live dashboards. De fato, você pode aplicar os algoritmos de aprendizado de máquina e processamento de gráficos do Spark nos fluxos de dados.



Para deixar claro a todos, **socket** é o elo de comunicação entre os processos do servidor e do cliente. Podemos dizer que é o caminho pelo qual os processos enviam e recebem mensagens.

A principal abstração do Spark Streaming é conhecida como fluxo discretizado (*Discretized Stream*), também conhecido como **DStream**. Os fluxos de dados de entrada são convertidos em DStream, que internamente criam uma **sequência de RDDs**. Isso permite uma **integração perfeita do Spark Streaming** com os componentes principais do Spark, MLlib, SQL DataFrames e GraphX.

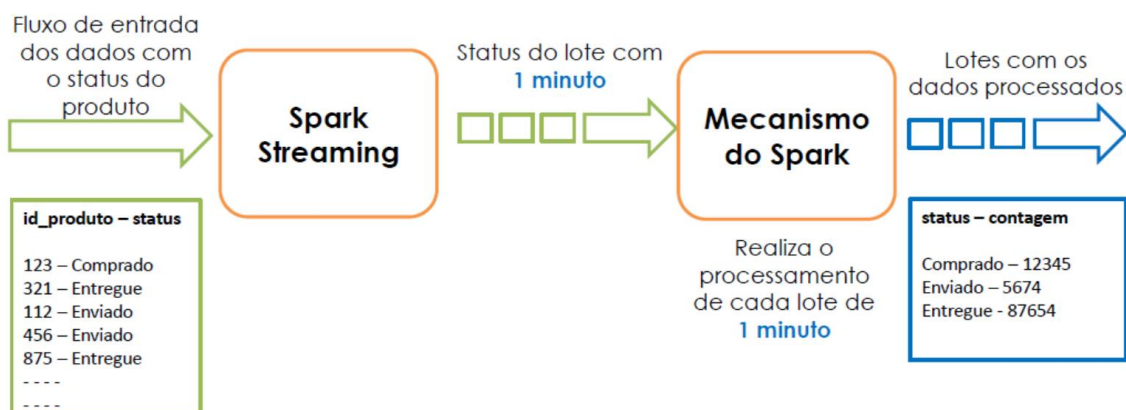
Um caso de exemplo onde pode-se utilizar Spark Streaming é para análise em Tempo Real. Vamos ao problema deste exemplo:

Problema: Construir um Dashboard informativo com a quantidade de produtos sendo:

- Comprados
- Enviados
- Entregues

A cada minuto!

O processo desenvolvido poderia ser assim:



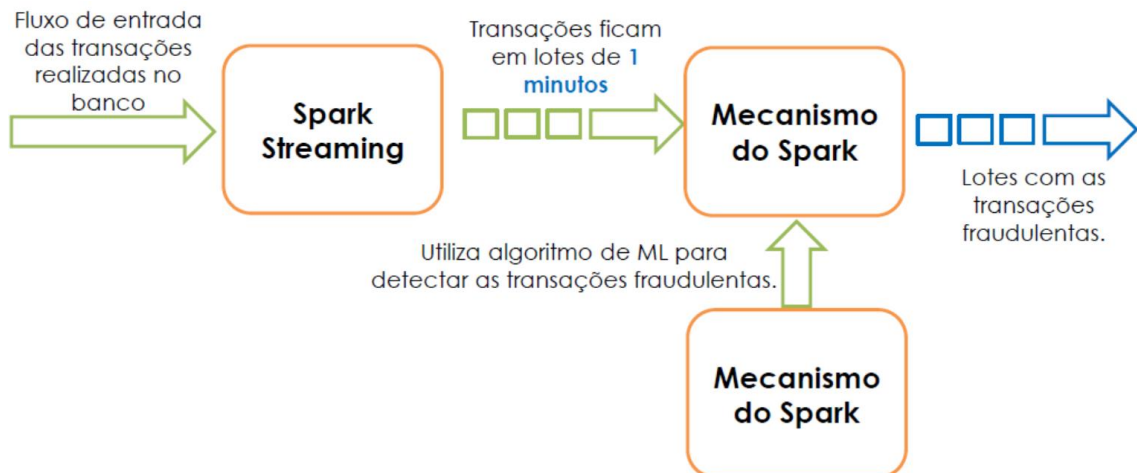
Outro caso de exemplo do uso de Spark Streaming pode ser em análise de sentimentos:

Problema: Construir um sistema de análise de sentimento em tempo real para verificar o sentimento de usuários de plataformas online (Mídias sociais, comentários em blogs e e-commerce, etc) a cada 15 minutos. O processo desenvolvido poderia ser assim:



Outro caso de exemplo do uso de Spark Streaming pode ser em detecção de Fraudes:

Problema: Construir um sistema de detecção de fraude para um banco com o objetivo de detectar transações fraudulentas.



A principal razão do Spark Streaming estar sendo adotado se deve ao **Apache Spark unificar todos os paradigmas de processamento de dados em um único framework**. Assim pode-se utilizar os módulos da forma que desejar, por exemplo combinando batch com streaming e com aprendizado de máquina. Desta forma é possível treinar modelos de aprendizado de máquina classificando dados provenientes do Streaming e realizando análises utilizando alguma ferramenta de analytics.

Em resumo, **DStream** é representado por um fluxo de dados divididos em pequenos lotes (micro batches). A API do Spark Streaming **transforma o fluxo de dados em micro lotes** e envia para o mecanismo de processamento do Spark. Assim, o Spark Streaming se beneficia do processamento em lote com fluxo (streaming). Ao invés de processar um evento por vez, um pequeno lote de eventos é processado. Isso **permite ao Spark manter um paradigma de programação unificado**, tanto para o processamento em lote ou em tempo real.

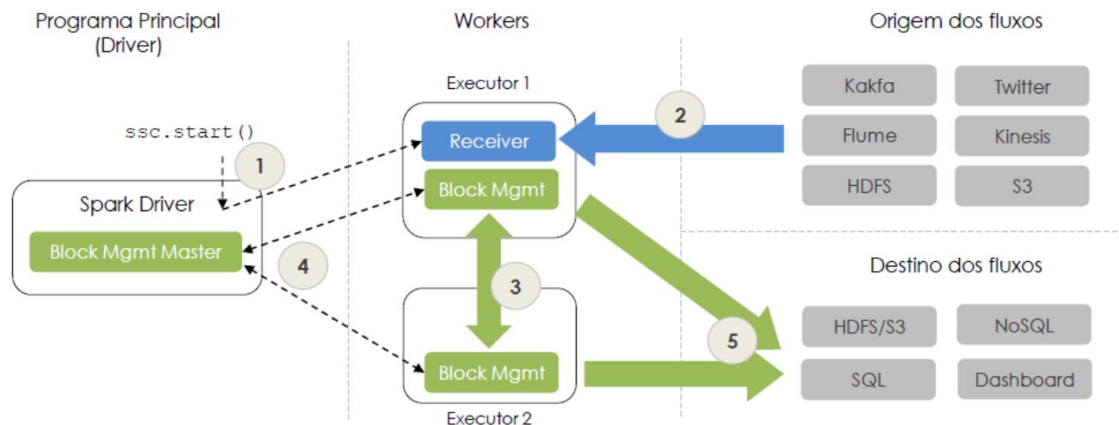
Como explicado anteriormente, no Spark Streaming, **um micro lote é criado baseado em tempo** (ao invés do tamanho), que são eventos recebidos em um intervalo de tempo (milissegundos, segundos ou minutos) e **agrupados em lote**. Isso garante que qualquer mensagem não demore

para ser processada, **mantendo a latência do processamento sob controle**. Outra vantagem do uso de micro lotes é manter o volume das mensagens de controle baixo. Por exemplo:

- Se um sistema necessita **que uma confirmação seja enviada pelo mecanismo de processamento**, então no caso de micro lotes, apenas uma confirmação é enviada por lote ao contrário se fosse por mensagem.
- Micro lotes também têm uma desvantagem, **pois em caso de falhas, todo o lote necessita ser reenviado**, mesmo que apenas uma mensagem no lote tenha falhado.

O fluxo de dados entre os componentes envolvidos no Spark.

1. Quando o contexto do Spark Streaming inicia, o programa principal (Driver) irá executar uma tarefa longa nos executores. O Spark Driver fica rodando para sempre, enquanto os executores têm a duração definida apenas para o processamento
2. O **Receiver** nos executores (Executor 1) **recebe os dados do fluxo de uma das origens**. Com o fluxo de dados de entrada, o Receiver divide o fluxo em blocos e mantém esses blocos em memória.
3. **Esses blocos são então replicados para outro executor** para evitar perda dos dados.
4. O ID do bloco então é transmitido para o Block Management Master no programa principal (Driver).
5. **Para cada intervalo do lote** configurado dentro do contexto do Spark, o programa principal irá **iniciar as tarefas Spark** para processar esses blocos. Esses blocos então são persistidos para qualquer armazenamento (S3, MySQL, NoSQL, HDFS, entre outros).



Programando com **DStreams** no Spark Streaming também segue um modelo similar, uma vez que o DStreams consiste em um ou mais RDDs. Quando os métodos como as transformações ou ações do Spark invocam o DStream, **uma operação equivalente é aplicada em todos os RDDs que constituem o DStream**. E nem todas as transformações e ações são suportadas em DStreams.

Por exemplo, assumindo que dado um intervalo de tempo em uma aplicação de processamento de dados no Spark Streaming, um DStream é gerado e consiste em múltiplos RDDs. Quando uma transformação é realizada no DStream, (por exemplo uma filtragem), **um novo DStream é gerado contendo apenas um RDD, dado a condição do filtro**.

Vamos executar um exemplo para melhor compreender o Spark Streaming. Podemos utilizar um simples código de contagem de palavras que foi apresentado na Sprint anterior com o objetivo

de entender o funcionamento. A ideia é entender os componentes envolvidos nessa aplicação e como os dados podem ser enviados e capturados dentro do Spark Streaming. Para simular a entrada dos dados, iremos utilizar um comando do Linux/Unix chamado nc.

- nc é um **utilitário que lê e escreve dados através das conexões de rede**. As palavras que iremos contar serão escritas em um socket TCP.

No caso do Windows, precisamos baixar uma ferramenta, neste exemplo usaremos o Netcat (<https://joncraton.org/blog/46/netcat-for-windows/>). Faça download e descompacte-o numa pasta. Através do Prompt de Comando navegue até a pasta e execute os comandos:

- 1) Para iniciar o serviço e abrir uma porta

```
c:\softwares\nc111nt>nc localhost 3456
```

- 2) O segundo para receber os comandos

```
c:\softwares\nc111nt>nc -L -p 3456
```

- 3) Agora basta digitar as mensagens para a porta

```
c:\softwares\nc111nt>nc localhost 3456  
  
c:\softwares\nc111nt>nc -L -p 3456  
teste  
teste2  
teste3
```

Para construir uma aplicação streaming sugere-se seguir os passos abaixo:

1. Criar o StreamingContext
2. Criar um DStream
3. Realizar as operações Dstream em paralelo em cada lote RDD criado
4. Iniciar o fluxo
5. Testar a aplicação

Vamos criando um **StreamingContext**. O principal ponto de entrada para criar aplicações Spark Streaming é o contexto de fluxo, chamado StreamingContext. Ele é configurado da mesma forma que o contexto do Spark, porém inclui um parâmetro adicional, **que é a duração do lote**, que pode ser em milissegundos, segundos ou minutos. É necessário ter um SparkContext disponível.

Primeiro, deve-se importar os contextos necessários.

```
In [1]: from pyspark import SparkContext  
        from pyspark.streaming import StreamingContext
```

Depois precisa-se criar o contexto Spark, neste exemplo usaremos 2 threads.

```
In [2]: sc = SparkContext("local[2]", "ContarPalavrasStreaming")
```

É importante notar que para testes locais deve-se utilizar, na configuração do Master, **pelo menos 2 threads**. Pois caso esteja utilizando, por exemplo Kafka, Flume ou Sockets, uma única thread será alocada para esses receptores, deixando os dados de entrada sem nenhum processo para ser utilizado.

Criar o contexto de Streaming com um intervalo de lote determinado. Nesse caso, estamos rodando micro lotes a cada 10 segundos.

```
In [3]: ssc = StreamingContext(sc, 10)
```

O segundo passo, é **criar um DStream** que irá conectar ao endereço localhost na porta 3456.

```
In [4]: dados = ssc.socketTextStream("localhost", 3456)
```

Os fluxos serão extraídos via `ssc.socketTextStream`, o qual é um método que revisa um fluxo de texto de um socket em particular. Nesse caso a própria máquina no socket 3456.

Uma vez que o DStream esteja disponível, podemos aplicar as **operações em paralelo** para cada lote de RDDs no fluxo. Podemos aplicar diversas transformações e ações no DStream, no caso de contar palavras, temos que realizar as seguintes transformações:

```
In [6]: # Dividir as linhas em palavras
palavras = dados.flatMap(lambda linha: linha.split(" "))
# Contar cada palavra em cada lote
pares = palavras.map(lambda palavra: (palavra, 1))
# Imprimir os primeiros 10 elementos de cada RDD neste DStream
contagem = pares.reduceByKey(lambda x, y: x + y)

contagem.pprint()
```

Para iniciar um fluxo do Spark Streaming e esperar por algum comando de execução:

```
In [*]: ssc.start()
        ssc.awaitTermination()
```

Para testar, basta digitar as mensagens no prompt de comando e então num loop de 10 segundos o Spark monta micro batches. Veja o resultado abaixo:

```
In [*]: ssc.start()
        ssc.awaitTermination()

-----
Time: 2019-10-29 10:44:40
-----

Time: 2019-10-29 10:44:50
-----

Time: 2019-10-29 10:45:00
-----

Time: 2019-10-29 10:45:10
-----
('teste3', 1)
('teste', 1)
('teste2', 1)
-----

Time: 2019-10-29 10:45:20
-----

Time: 2019-10-29 10:45:30
-----
```

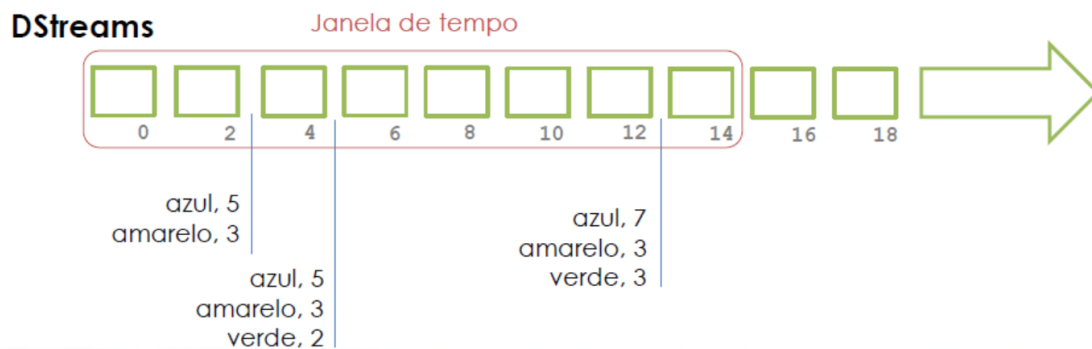
Recuperação de Falhas

Como comentado anteriormente, o programa Spark implementado realiza a contagem de palavras dentro do tempo determinado para o intervalo. Vamos supor que o programa Spark Streaming Python que criamos agora está definido para rodar a cada 2 segundos e pode retornar os seguintes valores nas respectivas marcas:

- 2 segundos, retornou o valor de {(azul, 5), (amarelo, 3)}.
- 4 segundos, retornou o valor de {(verde, 2)}.
- 12 segundos, retornou o valor de {(verde, 1), (azul, 2)}.

Porém, se quiséssemos o **valor agregado da contagem de palavras** em uma determinada janela de tempo? Digamos a cada 14 segundos.

Nesse caso, é preciso calcular a soma acumulada da contagem das palavras. Assim, na marca 4 segundos não teremos apenas o valor de {(verde, 2)}, mas sim o {(azul, 5), (amarelo, 3), (verde, 2)}



Para adicionar essa função de agregação em nosso exemplo, vamos criar um novo programa com base no código já desenvolvido. Para isso temos que:

- 1) Criar o contexto de Streaming
- 2) Criar um ponto de verificação
- 3) Criar uma função de atualização
- 4) Criar o DStream
- 5) Realizar as transformações e ações necessárias
- 6) Iniciar o fluxo

Começando com a criação do contexto de Streaming

```
In [1]: from pyspark import SparkContext
from pyspark.streaming import StreamingContext
sc = SparkContext("local[2]", "ContarPalavrasJanelaTempo")
ssc = StreamingContext(sc, 2)
```

Como uma aplicação de streaming deve funcionar 24/7, ela deve ser tolerante a falhas não relacionadas a lógica da aplicação. Por exemplo, falhas no sistema, na JVM, etc.). Dessa forma, se **um nó falhar é possível reagendar as tarefas** em outros Workers. Para isso, o Spark Streaming necessita criar um **ponto de verificação** (checkpoint) com informações suficientes em um sistema de armazenamento com tolerância a falhas (por exemplo, o HDFS). Existem dois tipos de dados em que podemos criar pontos de verificação:

- Ponto de verificação dos metadados
- Ponto de verificação dos dados

Ponto de verificação dos metadados: É utilizado para recuperar falhas de um nó rodando a aplicação principal (driver) de Streaming. O metadados inclui:

- **Configuração:** a configuração que foi utilizada para criar a aplicação de streaming.

- **Operações DStream:** o conjunto de operações DStream que definem a aplicação de Streaming.
- **Lotes incompletos:** lotes em que os Jobs são enfileirados, porém ainda não foram completados.

Ponto de verificação dos dados:

- Salvar os RDDs gerados para armazenamento confiável.
- É necessário em algumas transformações com estado que combinam dados em vários lotes.
- Pois elas dependem de lotes anteriores de RDDs, o que faz com que o comprimento da cadeia de dependência continue aumentando com o tempo.
- Para evitar que esses aumentos ilimitados no tempo de recuperação (proporcional a cadeia de dependência), periodicamente, são criados pontos de verificação (checkpoint) em armazenamento confiável para armazenar os RDDs intermediários das transformações com estado, eliminando as cadeias de dependência.

Quando habilitar o ponto de verificação (checkpoint)?

- Uso de transformações com estado: se estiver utilizando `updateStateByKey` ou `reduceByKeyAndWindow`, então deve-se criar um diretório para ser utilizado para criar os pontos de verificação periódico dos RDDs.
- Recuperar de falhas do programa principal (driver) rodando a aplicação: o ponto de verificação dos metadados são utilizados para recuperar a informação do progresso.

Para criar o checkpoint, precisa-se definir o diretório onde a informação será armazenada.

```
In [2]: ssc.checkpoint("checkpoint")
```

Para adicionar uma **função de agregação** neste exemplo, vamos criar uma função. A função de atualização diz ao programa para atualizar o estado da aplicação via `updateStateByKey`. Nesse caso é retornado a soma dos valores anteriores com os novos valores.

```
In [3]: def funcaoAtualizar(novos_valores, ultimos_valores):  
        return sum(novos_valores) + (ultimos_valores or 0)
```

Como fizemos anteriormente, para criar o DStream basta utilizar o `socketTextStream`.

```
In [4]: dados = ssc.socketTextStream("localhost", 3456)
```

Até agora vimos as transformações que atuam em lotes individuais sem a necessidade de saber o estado do lote anterior, que são chamadas de **Transformações Stateless**. Além desse tipo de transformações, existem também as **Transformações Stateful**, onde os resultados do lote anterior são utilizados para computar os resultados do lote atual. Esse tipo de transformações atua dentro de uma **janela deslizando** de períodos de tempo. O uso do **checkpoint** é importante e necessário para habilitar esse tipo de transformação.

```
In [5]: # Calcula a contagem  
palavras = dados.flatMap(lambda linha: linha.split(" "))  
  
pares = palavras.map(lambda palavra: (palavra, 1))  
  
contagem = pares.updateStateByKey(funcaoAtualizar)  
  
# Imprimir os primeiros 10 elementos de cada RDD neste DStream  
contagem.pprint()
```

Iniciar o fluxo de coleta via streaming

```
In [*]: ssc.start()  
ssc.awaitTermination()
```

Inserindo a sequência abaixo, obteve-se o seguinte resultado:

```
azul  
azul  
azul  
azul  
azul  
verde  
verde  
verde  
verde  
branco  
branco  
branco  
preto  
preto  
preto  
preto  
verde  
azul  
azul  
verde  
preto
```

Começou assim:

```
In [*]: ssc.start()  
ssc.awaitTermination()
```

```
Time: 2019-10-29 14:19:52
```

```
('azul', 4)
```

```
Time: 2019-10-29 14:19:54
```

```
('azul', 4)
```

```
Time: 2019-10-29 14:19:56
```

```
('azul', 5)
```

```
In [*]: ssc.start()  
ssc.awaitTermination()  
('branco', 2)
```

```
Time: 2019-10-29 14:20:06
```

```
('verde', 4)
```

```
('azul', 5)
```

```
('branco', 2)
```

```
Time: 2019-10-29 14:20:08
```

```
('verde', 4)
```

```
('azul', 5)
```

```
('branco', 3)
```

Terminou assim:

```
In [*]: ssc.start()
       ssc.awaitTermination()
('branco', 3)
('preto', 4)
```

```
-----
Time: 2019-10-29 14:20:44
-----
```

```
('verde', 6)
('azul', 7)
('branco', 3)
('preto', 4)
```

```
-----
Time: 2019-10-29 14:20:46
-----
```

```
('verde', 6)
('azul', 7)
('branco', 3)
('preto', 5)
```

Recuperação de falhas

As aplicações de Streaming, geralmente, são compostas por diversos componentes e podem acontecer diferentes tipos de falhas. Para melhor entender as semânticas fornecidas pelo Spark Streaming, vamos relembrar o básico sobre **tolerância a falhas do Spark RDDs**:

- Um RDD é **imutável**, deterministicamente recomputável e distribuído.
- Se alguma partição do RDD for perdida devido a alguma falha no nó Worker, então a partição pode ser **recomputada do conjunto de dados original utilizando a linhagem das operações**.
- Assumindo que todas as transformações em RDD são determinísticas, o dado no RDD final transformado, sempre será o mesmo, independentemente das falhas que ocorrem.

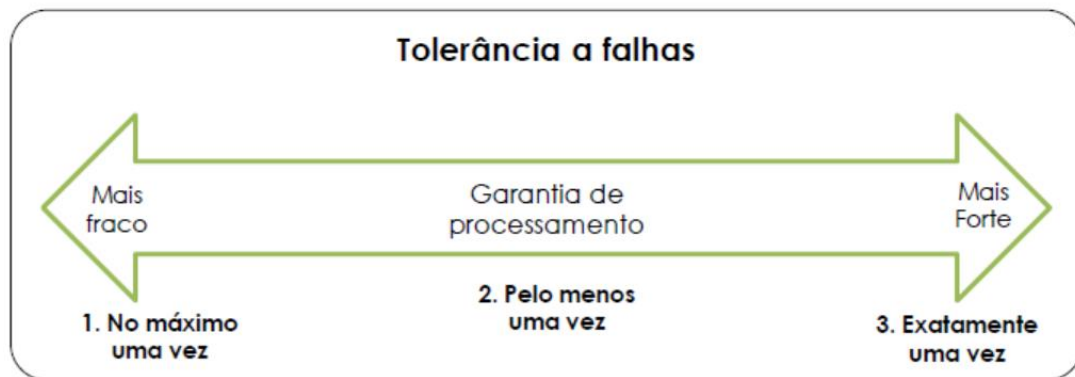
Spark opera com os dados em sistemas de arquivos tolerante a falhas, como HDFS e S3. No entanto, este não é o caso do Spark Streaming, pois os dados na maioria das vezes são recebidos via rede. Para obter as mesmas propriedades de tolerância a falhas para todos os RDDs gerados, **os dados recebidos são replicados entre vários executores Spark em diferentes Worker**. Isso leva a dois tipos de dados no sistema que precisam ser recuperados em caso de falhas:

- Dados recebidos e replicados;
- Dados recebidos, mas armazenados em buffer para replicação.

No primeiro caso (Dados recebidos e replicados) o dado sobrevive a falha de um único Worker, uma vez que uma cópia existe em um dos outros nós. Já no segundo caso (Dados recebidos, mas armazenados em buffer para replicação), como o dado não foi replicado, a única forma de recuperar esses dados é acessá-los novamente na origem. Além disso, temos outras duas falhas que podem ocorrer:

- Falha no Worker: Qualquer um dos nós Worker que rodam os executores podem falhar e todos os dados que estão em memória serão perdidos. Se algum receptor estiver executando em nós com falhas, então os dados armazenados serão perdidos.
- Falha no Driver: Se o nó que está rodando o programa principal (Driver Program) falhar, então perderemos o SparkContext, e todos os executores com dados em memória serão perdidos.

Nas aplicações de Streaming existem tipicamente três tipos de garantias que um sistema pode fornecer em todas as condições de operação possíveis:



1.No máximo uma vez (*at most once*): Cada registro será processado uma vez ou não será processado.

2.Pelo menos uma vez (*at least once*): Cada registro será processado uma ou mais vezes. Esse é mais forte que o anterior, pois garante que nenhum dado seja perdido, porém **pode existir duplicações**.

3.Exatamente uma vez (*exactly once*): Cada registro será processado exatamente uma vez. Esta é a semântica que tem a garantia mais forte, uma vez que nenhum dado será perdido e nenhum dado será processado várias vezes.

Basicamente em qualquer sistema de processamento de fluxos (streaming) existem três passos para processar os dados:

1.Recebendo os dados: os dados são recebidos de fontes utilizando os receptores ou de alguma outra forma.

2.Transformando os dados: os dados recebidos são transformados usando as transformações DStream/RDDs.

3.Salvando os dados: os dados que foram transformados são enviados para sistemas externos (sistema de arquivos, banco de dados, Dashboards) para serem persistidos/visualizados.

Se uma aplicação de Streaming precisa alcançar as garantias exatamente uma vez (*exactly-once*) fim-a-fim, será necessário que cada passo do processamento dos fluxos **também apresentem a garantia de exatamente uma vez**. Ou seja, cada registro deve ser recebido exatamente uma vez, transformado exatamente uma vez e persistido exatamente uma vez. Vamos entender a semântica dessas etapas no contexto do Spark Streaming:

Recebendo os dados: Diferentes origens de dados fornecem diferentes garantias.

- **Com arquivos:** se todos os dados estiverem em um sistema tolerante a falhas então o Spark Streaming pode recuperar de qualquer falha e processar novamente os dados. Isso permite a semântica exatamente uma vez.
- **Com origens baseadas em receptores:** Irá depender do cenário de falha e do tipo do receptor:
 - **Receptor Confiável:** Esses receptores enviam a confirmação depois de garantir que os dados recebidos foram replicados. Portanto, se o receptor for reiniciado, a origem irá reenviar os dados e nenhum dado será perdido devido à falha.
 - **Receptor Não Confiável:** Esses receptores **NÃO** enviam a confirmação e portanto podem perder dados quando uma falha ocorrer no Driver ou Worker.

- **Com API direta do Kafka:** faz parte da versão 1.3+ do Spark e garante que todos os dados Kafka sejam recebidos pelo Spark Streaming exatamente uma vez. Se implementar a operação de saída exatamente uma vez, é possível atingir a garantia exatamente uma vez fim-a-fim.

Salvando os dados: as operações de saída (como foreachRDD) tem a semântica pelo menos uma vez, ou seja, o dado transformado pode ser escrito mais de uma vez caso ocorra alguma falha no Worker. Enquanto isso é aceitável para salvar em sistemas de arquivos utilizando as operações de saveAs***Files, será necessário um pouco mais de trabalho para conseguir atingir a semântica de exatamente uma vez. Existem duas abordagens:

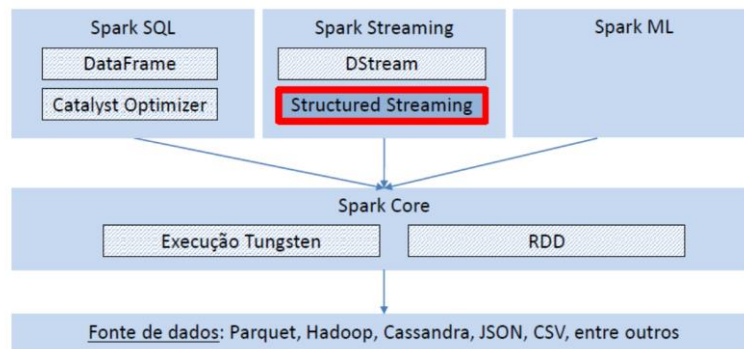
- Atualizações consecutivas: Várias tentativas de escrever sempre os mesmos dados. Por exemplo saveAs***Files sempre irá gravar os mesmos dados nos arquivos que foram gerados.
- Atualizações transacionais: todas as atualizações são feitas de forma transacional para que as atualizações sejam feitas exatamente uma vez. Uma forma de conseguir isso seria:
 - Utilize o tempo do lote (disponível em foreachRDD) e o índice de partição do RDD para criar um identificador. Esse identificador irá identificar de forma exclusiva o dado na aplicação Streaming.
 - Atualizar o sistema externo com esse dado de maneira transacional utilizando o identificador. Ou seja, se o identificar ainda não foi enviado, deve-se enviar os dados da partição e o identificador. Caso contrário, ele já existe e não necessita de atualização.



Spark Streaming Estruturado

Como vimos, o Apache Spark migrou o processamento de dados em RDD para um processamento estruturado (via DataFrame/Datasets). A API Dataframe permite a possibilidade de realizar as **otimizações de desempenho do Tungsten e do Catalyst de maneira mais eficiente**. A partir da versão 2.2 do Spark o Streaming Estruturado foi marcado como estável.

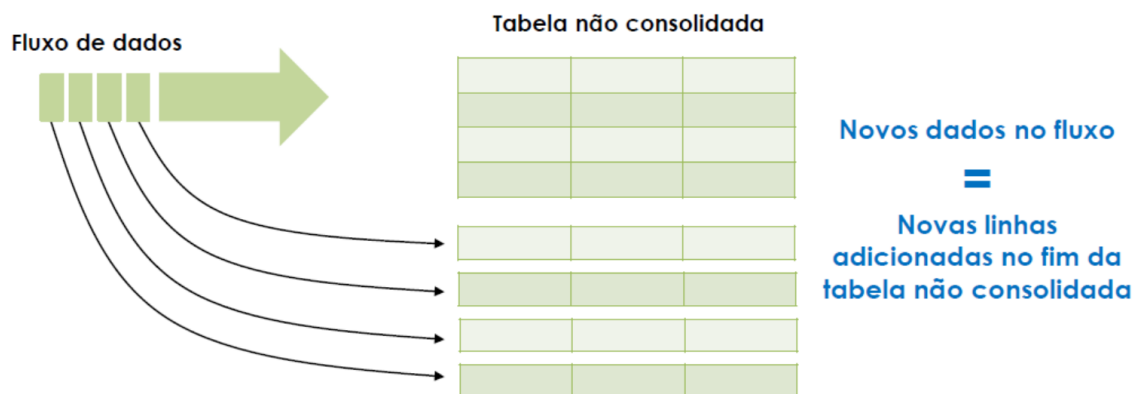
Spark Streaming



As aplicações de Streaming tendem a crescer na complexidade. As computações em fluxo **não** rodam de maneira isolada, **elas interagem com os sistemas de armazenamento, aplicações em lote e bibliotecas de aprendizagem de máquina**. Portanto, as aplicações contínuas nunca param e continuam a produzir dados à medida que novos dados chegam. Com a API de Streaming Estruturado podemos realizar as mesmas operações que realizamos na API DataFrame. Permite extrair valor de sistemas de fluxos de maneira rápida, **sem alterar o código previamente desenvolvido**. Podemos criar protótipos das aplicações e então convertê-las para um job de streaming. E isso é feito pelo processamento incremental dos dados.

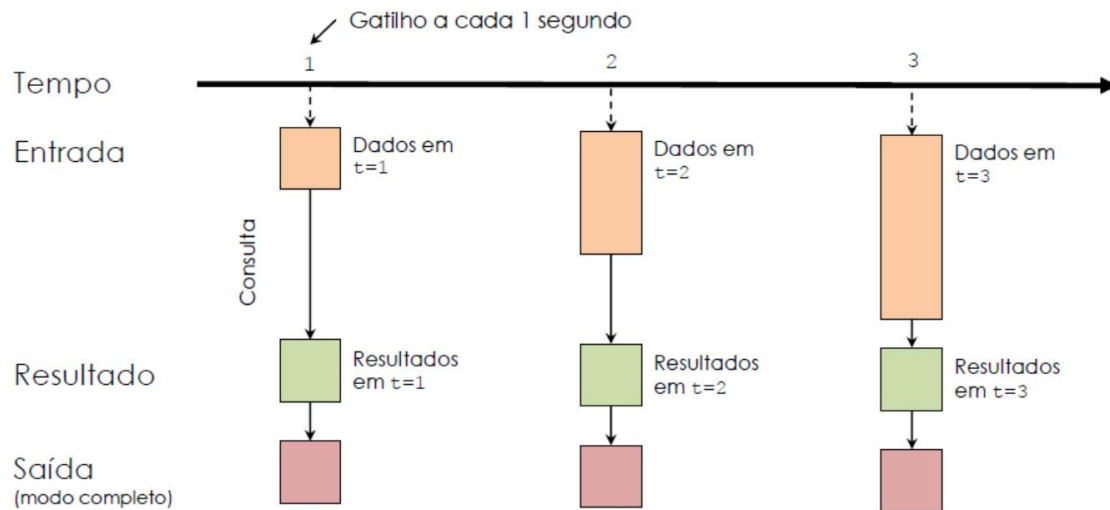
Como vimos uma aplicação contínua também pode ser implementada em cima dos RDDs e DStreams **porém é necessário o uso de duas diferentes APIs**.

No **Spark Structured Streaming** as APIs foram unificadas. Essa unificação é alcançada ao visualizar o **streaming estruturado como uma tabela relacional sem limites**, onde novos dados são anexados continuamente ao final da tabela.



Uma consulta de entrada irá gerar uma **"tabela de resultado"**. Cada intervalo de gatilho (digamos, a cada 1 segundo), novas linhas são anexadas a tabela de entrada, que eventualmente

irá atualizar a Tabela de Resultados. Sempre que a tabela de resultados for atualizada, iremos escrever as alterações das linhas em uma área externa (**external sink**).



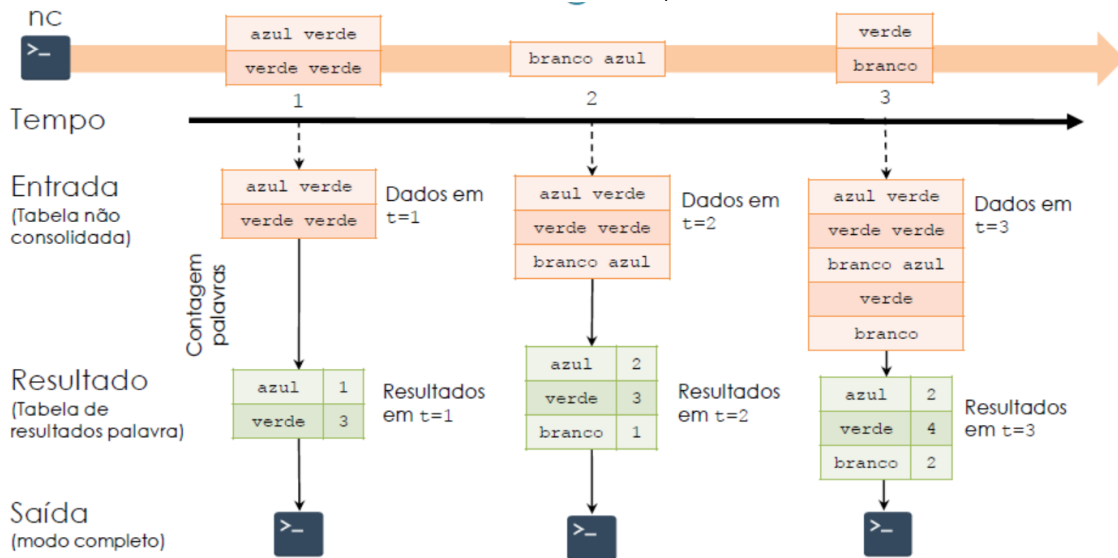
A saída é definida como o que é escrito no armazenamento externo. A saída pode ser definida em modos diferentes:

- **Modo completo** (complete): A tabela de resultado inteira será escrita em um armazenamento externo. É o conector do armazenamento que irá decidir como será tratado a escrita de toda a tabela.
- **Modo anexo** (append): Apenas as novas linhas que foram anexas na tabela de resultado desde o último gatilho serão escritas no armazenamento externo. Isso só é aplicado nas consultas onde as linhas existentes na tabela de resultado não mudam.
- **Modo atualização** (update): Apenas as linhas que foram atualizadas na tabela de resultado desde o último gatilho serão escritas no armazenamento externo (Spark 2.1.1+).

Para ilustrar os conceitos vistos nessa seção, iremos modificar o exemplo de contagem de palavras que vimos via DStream para Streaming Estruturado.

Como estamos utilizando a API Streaming Estruturado, temos que **aprender novos conceitos** de programação e seguir os seguintes passos:

- Importar as bibliotecas necessárias.
- Criar o contexto do Spark.
- Criar uma sessão para trabalhar com a API Estruturada
- Criar o fluxo que irá gerar o DataFrame
- Realizar a contagem das palavras
- Iniciar a consulta para imprimir o resultado no terminal
- Criar código para esperar a finalização do fluxo.



Iremos utilizar o **SparkSession** e duas funções do módulo `pyspark.sql` que serão utilizadas para realizar as transformações nos dados. Além dessas funções, iremos utilizar diretamente o `SparkSession` para realizar a leitura do fluxo em um determinado Socket. Também podemos criar o contexto do Spark para realizar algumas configurações via código.

```
In [1]: from pyspark import SparkContext
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split
sc = SparkContext()
sc.setLogLevel('ERROR')
```

Os parâmetros para criar uma sessão Spark são:

```
In [3]: spark = SparkSession \
.builder \
.appName("ContarPalavrasEstruturado") \
.getOrCreate()
```

Iremos utilizar a método `readStream` da API Estruturada que irá criar o `DataFrame` que representa o fluxo de entrada de linhas da conexão via nc. O caractere “\” serve para quebrar linha do código. Se preferir pode digitar tudo em uma única linha.

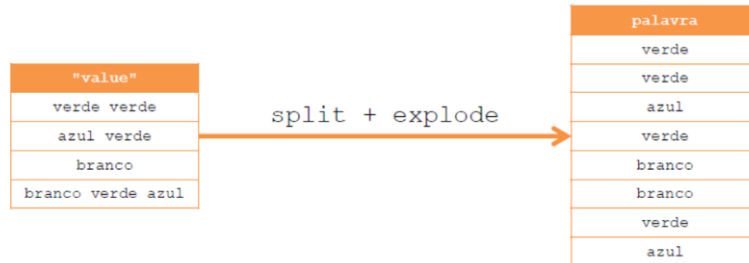
```
In [4]: linhas = spark.readStream\
.format("socket")\
.option("host", "localhost")\
.option("port", 3456)\
.load()
```

O `DataFrame` `linhas` representa uma tabela não consolidada que contém os dados do texto do Streaming. Essa tabela contém apenas uma coluna de strings nomeada de “**value**”, e cada linha nos dados do texto de Streaming se tornam linhas na tabela. O próximo passo é dividir cada linha em múltiplas linhas com uma palavra cada.

As funções:

- **explode:** Retorna uma nova linha para cada elemento em um mapeamento ou array.
- **split:** Divide uma string em um determinado padrão.


```
In [5]: palavras = linhas.select(
        explode(
            split(linhas.value, " ")
        ).alias("palavra")
    )
```



Por fim, realiza-se a contagem de palavras agrupando a coluna 'palavra' que foi criado no passo anterior.

```
In [6]: contagem = palavras.groupBy('palavra').count()
```

Para iniciar a consulta que queremos temos que escrever o fluxo de saída, qual o modo, bem como o formato.

```
In [7]: consulta = contagem.writeStream\
        .outputMode("complete")\
        .format("console")\
        .start()
```

A saída da contagem deve ser realizada para imprimir o conjunto completo de contagens (especificado pelo outputmode("complete")) no terminal **quando ele for atualizado**.

Espera finalizar a consulta através de uma finalização forçada.

```
In [*]: consulta.awaitTermination()
```

Processo vai ficar rodando e um conjunto de palavras pode ser adicionado

```
c:\softwares\nc111nt>nc localhost 3456

c:\softwares\nc111nt>nc -L -p 3456
verde
verde
verde
preto
preto
branco
branco
preto
azul
```

A console do jupyter vai retornar o resultado

```
[I 13:46:28.627 NotebookApp] Saving file at /ApacheSpark-StreamingEstruturado.ipynb
[Stage 1:=====] (170 + 8) / 200[I 13:48:28.628 NotebookApp] Saving file at /ApacheSpark-StreamingEstruturado.ipynb
Batch: 0
-----
|palavra|count|
-----
[Stage 3:=====] (88 + 8) / 200[I 13:50:28.646 NotebookApp] Saving file at /ApacheSpark-StreamingEstruturado.ipynb
Batch: 1
-----
|palavra|count|
| verde| 1|
-----
Batch: 2
-----
|palavra|count|
| preto| 3|
| verde| 3|
| branco| 2|
| azul| 1|
-----
```

Exercícios:

- 1) Modifique esse código que conta palavras modificando o modo de saída para **Modo Completo** (complete)
- 2) Modifique esse código que conta palavras modificando o modo de saída para **Modo anexo** (append)
- 3) Modifique esse código que conta palavras modificando o modo de saída para **Modo atualização** (update)

Além do modo de saída, existem locais ("sinks") onde podemos enviar os dados[4]:

- **File Sink** –Armazena a saída em um diretório.
- **Kafka Sink** –Armazena a saída em um ou mais tópicos Kafka.
- **Foreach Sink** –Roda computações em cada um dos registros de saída.
- **Memory Sink** –A saída é armazenada em memória como se fosse uma tabela em memória (apenas para debug).
- **Console Sink** –A saída é impressa no console toda vez que um gatilho (trigger) acontece.

Os **Triggers**, ou Gatilhos em português, definem quando uma consulta (query) é processada, se ela será executada como *micro-batch* com um intervalo de tempo fixado ou como sendo processada de forma contínua.

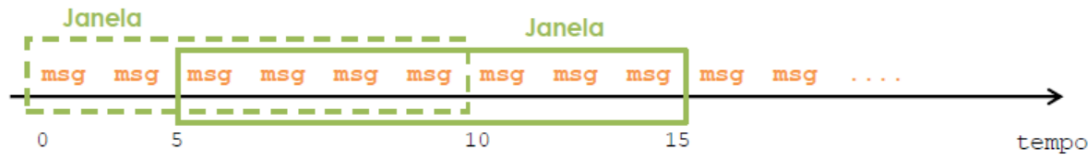
- Não especificado (padrão)
- Micro-lotes com Intervalos fixos
- Micro-lote uma única vez
- **Contínuo** com intervalo fixo de checkpoint

Windowing

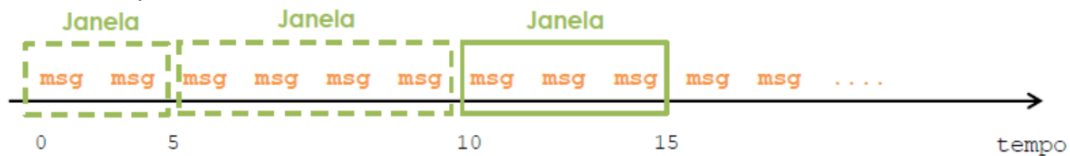
Janelas especificam a granularidade ou número de registros subsequentes, que são utilizados para executar funções de agregação nos fluxos. Existem basicamente **5 diferentes propriedades** em **duas dimensões** em que a Janela pode ser definida, onde cada definição da janela precisa usar uma propriedade de cada dimensão.

1. A primeira dimensão é o modo em que as janelas subsequentes de um fluxo contínuo de tuplas podem ser criadas: **janela deslizante** (sliding window) e **janela giratória** (tumbling window).

Janela deslizantes – remove uma tupla de elementos sempre que uma nova tupla é elegível para ser incluída.



Janela giratória – remove todas as tuplas quando existe tuplas suficientes chegando para criar uma nova tupla.



2. A segunda é que o número de tuplas que caem em uma Janela devem ser especificadas: seja baseado em **contagem**, **tempo** ou **sessão**.

Janela baseada na contagem – Esse tipo de propriedade de janela sempre terá os **n elementos mais novos**. Isso pode ser realizado via política de atualização da janela deslizando ou giratória.

Janela baseada na sessão – Utiliza o ID de sessão de uma tupla para determinar se ela pertence a alguma janela. Normalmente, contém todos os dados de uma interação de usuário, por exemplo, em uma compra online.

Janela baseada no tempo – Essa janela utiliza uma tupla de **timestamp** para determinar se ela pertence a alguma janela. Esse tipo de janela pode ser utilizado para dados que chegam atrasado, que é um conceito bem interessante do Apache Spark Streaming Estruturado. É importante notar que o número de tuplas por janela pode ser diferente uma vez que depende apenas em quantas mensagens em um determinado período de tempo chegaram.



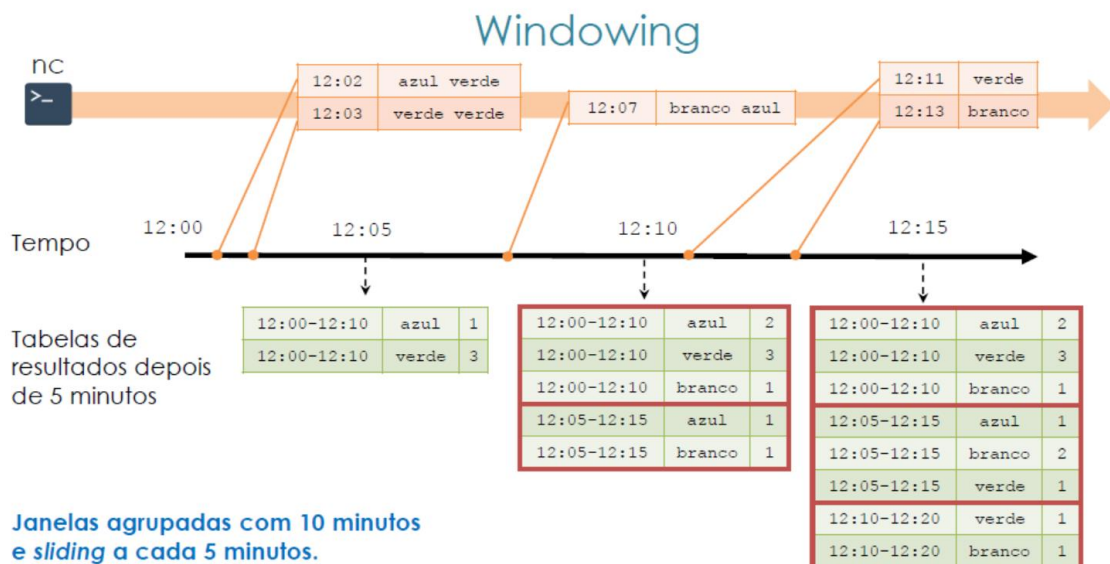
Apache Spark Streaming Estrutura é **mais flexível no modelo de processamento em janela**. Uma vez que os fluxos são tratados com tabelas contínuas sendo adicionadas ao conjunto. Como cada linha na tabela tem um **timestamp**, as operações em janelas podem ser especificadas na consulta e cada consulta pode ser diferente em cada janela. De certa forma, esse processo é um tipo especial de **ordenação de agrupamento baseado na coluna de timestamp**. Isso facilita o tratamento dos dados que chegam em atraso, pois o Spark pode inclui-lo na janela apropriada e realizar a recomputação na janela em que o dado foi adicionado com atraso. Essa é uma característica altamente configurável.

Em análise de series temporais e especialmente na computação de fluxos, cada registro é atribuído a um timestamp particular. Uma forma de criar esse timestamp é no **momento de**

chegada nos mecanismos de processamento do fluxo. Porém, é possível criar um evento de tempo para cada registro em um momento particular na linha do tempo. Por exemplo, quando uma medição de um dispositivo qualquer de IoT chega. Isso permite **lidar com a latência entre a criação e o processamento de um evento**. Por exemplo, se o sensor de IoT estiver desligado durante um grande período de tempo ou a rede está congestionada, irá causar um atraso na entrega dos dados. Para melhor ilustrar esse conceito iremos aplicar operações em janela em um determinado evento. Vamos voltar ao exemplo de contagem de palavras, e modificar o fluxo para ao invés de contar palavras toda vez que uma nova for inserida, iremos contar as palavras durante **um determinado intervalo de tempo**, por exemplo 10 minutos.

Para isso temos que:

- Importar uma nova biblioteca.
- Modificar o fluxo que irá gerar o DataFrame para incluir o timestamp.
- Modificar a contagem das palavras para incluir o timestamp de cada linha.
- Realizar a contagem por janela.
- Realizar a consulta.



No começo do arquivo adicione essa nova linha:

```
from pyspark.sql.functions import window
```

No fluxo que estamos criando temos que adicionar uma nova opção para adicionar o timestamp para cada novo fluxo de entrada. Para isso, adicione a seguinte opção na parte do código que cria o fluxo e salva no DataFrame linhas.

```
.option("includeTimestamp", "true")
```

Na consulta iremos selecionar as palavras, bem como o timestamp de gerado.

```
palavras = linhas.select(
    explode(split(linhas.value, " ")).alias("palavra"),
    linhas.timestamp
)
```

O agrupamento realizado será feito pela janela (window) de 10 minutos e sliding (deslizamento) a cada 5 minutos. Para isso iremos utilizar a função window que existe no módulo sql do pyspark.


```
contagem = palavras.groupBy(
    window(palavras.timestamp, "10 minutes", "5 minutes"),
    palavras.palavra
).count().orderBy('window')
```

Por fim, adicione a seguinte opção para visualizar melhor a saída na consulta.

```
.option("truncate", "false")
```

Exercício: Crie essa aplicação explicada acima com o nome da aplicação como “exercicio_janela”

A imagem abaixo ilustra uma comparação com diversos mecanismos de streaming.



Property	Structured Streaming	Spark Streaming	Apache Storm	Apache Flink	Kafka Streams	Google Dataflow
Streaming API	incrementalize batch queries	integrates with batch	separate from batch	separate from batch	separate from batch	integrates with batch
Prefix Integrity Guarantee	✓	✓	✗	✗	✗	✗
Internal Processing	exactly once	exactly once	at least once	exactly once	at least once	exactly once
Transactional Sources/Sinks	✓	some	some	some	✗	✗
Interactive Queries	✓	✓	✗	✗	✗	✗
Joins with Static Data	✓	✓	✗	✗	✗	✗

Desafio XPTO: Twitter

Seguindo o desafio planejado precisamos fazer nesse momento a Ingestão dos dados do Twitter. Antes vamos apresentar o Twitter.

Sobre o Twitter

O Twitter foi criado em março de 2006 por Jack Dorsey, Evan Williams, Biz Stone e Noah Glass e foi lançado em Julho de 2006 nos EUA. A ideia inicial dos fundadores era que o Twitter fosse uma espécie de "SMS da internet" com a limitação de caracteres de uma mensagem de celular. Inicialmente chamada Twttr (sem vogais), o nome da rede social, em inglês, significa gorjear. A ideia é que o usuário da rede social está "piando" pela internet [8].

No dia 12 de setembro de 2013, por meio do perfil da empresa no próprio Twitter, foi informado que ela havia enviado à SEC (CVM dos EUA) documentos confidenciais para sua abertura de capital na Bolsa de Valores, operação também conhecida como IPO (Oferta Pública Inicial, em inglês). No dia 7 de novembro de 2013, o Twitter fez sua estreia na Bolsa de Nova York. Todas as 70 milhões de ações colocadas no mercado foram vendidas. Seu valor chegou a subir até 90% de alta em relação ao valor estipulado inicialmente na abertura do pregão. Na ocasião, a empresa captou US\$1,82 bilhão no mercado e foi avaliada em US\$24,57 bilhões.

No dia 23 de janeiro de 2015 duas novidades foram inseridas. O "Enquanto você estava fora" é um resumo das principais notícias e/ou tweets de quem você segue. Já o "Digits" é feito para desenvolvedores e vai auxiliar no acesso a sites pelo celular. Ainda em 2015, o Twitter anunciou que irá liberar as mensagens diretas entre pessoas que não se seguem. Para isso, o usuário terá que liberar o outro contato por meio das configurações da conta.

O Twitter é o lugar para saber sobre o que está acontecendo no mundo agora e sobre o que as pessoas estão conversando [7]. O Twitter é uma rede social e um servidor para microblogging, que permite aos usuários enviar e receber atualizações pessoais de outros contatos (em textos de até 280 caracteres, conhecidos como "tweets"), por meio do website do serviço, por SMS e por softwares específicos de gerenciamento[8]. As atualizações são exibidas no perfil de um usuário em tempo real e também enviadas a outros usuários seguidores que tenham assinado para recebê-las. As atualizações de um perfil ocorrem por meio do site do Twitter, por RSS, por SMS ou programa especializado para gerenciamento. O serviço é gratuito pela internet, entretanto, usando o recurso de SMS pode ocorrer a cobrança pela operadora telefônica[8].

Você pode acessar o Twitter pela Web ou por dispositivo móvel. Para compartilhar informações no Twitter da forma mais ampla possível, empresas, desenvolvedores e usuários podem acessar os dados do Twitter de forma programática através de APIs (interfaces de programação de aplicativo). Este artigo explica o que são as APIs do Twitter, que tipo de informação fica disponível com elas e algumas medidas de proteção que o Twitter preparou para que as APIs sejam usadas[7].

Em última instância, as APIs são a forma como os programas de computador "conversam" entre si para trocar informações. Isso é feito permitindo-se a um aplicativo de software acessar um dispositivo conhecido como terminal: um endereço que corresponde a um tipo específico de informação que fornecemos (terminais são, geralmente, únicos, como números de telefone). Damos acesso a partes do nosso serviço através das APIs para permitir que os

desenvolvedores criem softwares que se integrem ao Twitter, como, por exemplo, uma solução que ajude uma empresa a medir opiniões dos clientes no Twitter[7].

Os dados do Twitter têm um caráter único de compartilhamento em relação a outras mídias sociais porque refletem as informações que os usuários escolheram compartilhar publicamente.

A plataforma de APIs do Twitter permite amplo acesso aos dados públicos do Twitter que os próprios usuários escolheram compartilhar com o mundo. Também há possibilidade de usar APIs que permitem aos usuários gerenciar suas informações privadas (ex.: Mensagens Diretas) e as compartilhar com os desenvolvedores que eles mesmos autorizaram.

Acesso aos seus dados do Twitter

Quando alguém deseja acessar as APIs do Twitter, há a solicitação de registro de um **aplicativo**. Por padrão, aplicativos só podem acessar informações públicas no Twitter. Determinados terminais, como aqueles responsáveis por enviar e receber Mensagens Diretas, necessitam que você dê permissões adicionais para que dados do usuário sejam acessados. **Essas permissões não são dadas por padrão**: o usuário decide se cada aplicativo poderá ter esse acesso e pode controlar todos os aplicativos [autorizados em sua conta](#).

As APIs do Twitter incluem uma grande variedade de terminais, que se dividem em cinco grupos principais:

Contas e usuários: o Twitter permite que desenvolvedores gerenciem, através dos programas, perfis e configurações de conta, silenciem usuários, administrem usuários e seguidores, solicitem informações sobre a atividade de uma conta autorizada. Esses terminais podem ajudar serviços públicos como por exemplo o Controle de Emergências do Departamento de Virgínia da Comunidade Britânica, que fornece aos moradores informações sobre atuações e alertas de emergência.

Tweets e respostas: Tornam tweets e respostas disponíveis aos desenvolvedores e permitindo que eles postem Tweets através de API. Os desenvolvedores podem acessar Tweets procurando por palavras-chave específicas ou solicitando uma amostra de Tweets de uma conta específica.

Esses terminais são utilizados por instituições, como a ONU por exemplo, para identificar, entender e combater a desinformação que afeta iniciativas de saúde pública. Na Indonésia, por exemplo, havia rumores persistentes de que as vacinas contêm produtos à base de suínos ou causam infertilidade. Entender como esses rumores se iniciavam e se espalhavam permitiu à ONU montar uma equipe de campo para desmentir essa informação, o que causou uma preocupação efetiva neste país de maioria muçulmana.

Mensagens Diretas: Os terminais de Mensagem Direta (MD) fornecem acesso a conversas de MDs de usuários que explicitamente permitiram esse acesso a aplicativos específicos. **O Twitter não comercializa Mensagens Diretas**. As APIs de Mensagens Diretas fornecem acesso limitado a desenvolvedores para criar experiências de personalização, como por exemplo o criador de suporte ao cliente no evento Wendy March Madness. Para suas contas próprias ou gerenciadas, as empresas podem criar essas experiências de diálogo com o cliente baseadas em chatbots ou interação humana para atendimento ao consumidor, ações de marketing e engajamento com a marca.

Anúncios: O Twitter fornece uma suíte de APIs para permitir que desenvolvedores ajudem outras empresas a criar e gerenciar automaticamente campanhas publicitárias no Twitter. Os desenvolvedores podem utilizar Tweets públicos para identificar assuntos e temas de interesse, bem como fornecer a empresas ferramentas para veicular campanhas publicitárias que conversem com o público variado do Twitter.

Ferramentas de publisher e SDKs: O Twitter fornece ferramentas para desenvolvedores de softwares e publishers inserirem, em websites, timelines do Twitter, botões de compartilhamento e outros conteúdos do Twitter. Essas ferramentas permitem que marcas adicionem conversas ao vivo e públicas do Twitter para suas ações de internet, de forma a facilitar ao cliente o compartilhamento de informações e artigos a partir dos seus websites.

Saiba mais sobre as APIs e as características de cada terminal em [documentação de desenvolvedor](#).

Em todas as APIs e produtos de dados, o Twitter se responsabiliza por proteger seriamente as informações dos usuários. Mantemos políticas e processos rigorosos avaliando como os desenvolvedores estão usando dados do Twitter e restringindo o uso impróprio de informações. Quando percebem que o conhecimento de que algum desenvolvedor violou as políticas, agem com as medidas adequadas, que podem incluir suspensão e proibição do acesso aos produtos de APIs e dados do Twitter.

Para saber mais sobre as APIs do Twitter, visite o site developer.twitter.com e verifique [as políticas e acordos](#).

Exercícios:

- 4) Leia atentamente o que está proposto no laboratório: “Criando Processo de coleta do Twitter”. Após execute os passos propostos do Laboratório.

Desafio XPTO: Ingestão do Twitter E Processamento Streaming

Vamos usar a mesma EMR utilizada para carregar o histórico para fazer a execução do processo de ingestão dos dados do Twitter com Python e vamos usar o Spark Streaming para executar o processamento dos dados da RAW para a REF.

Dicas:

- Crie um Cluster somente com o nó master e com a menor configuração possível com Spark
- Instale a Lib tweepy no cluster como feito no Exercício do Twitter.
- Execute o código Python via linha de comando na console do cluster para fazer a ingestão do Twitter.
- Para rodar a carga streaming da RAW para a REF use o comando do Spark: `spark-submit`.

Glue

Vamos usar um Glue Job com Python para criar o processo de ingestão dos dados do Twitter e outro Glue Job com Spark para o processamento dos dados da RAW para a REF.

Dicas:

- Utilize um Glue Job com Python para fazer a ingestão do Twitter.
- Crie Role com Permissões de GLUE, S3 e CloudWatch
- Incluir Libs extra para adicionar: tweepy

Aqui um exemplo de como fazer:

```
#Carregar e instalar a lib:
import os
import site
from importlib import reload
from setuptools.command import easy_install
install_path = os.environ['GLUE_INSTALLATION']
easy_install.main( ["--install-dir", install_path, "tweepy"] )
reload(site)
# Carregar demais Libs:
import json
import boto3
import tweepy
...
```

<https://stackoverflow.com/questions/46329561/use-aws-glue-python-with-numpy-and-pandas-python-packages>

- Utilize um Glue Job com Spark para executar o processamento dos dados streaming da RAW para a REF.

Exercícios:

IMPORTANTE: A ingestão dos dados do Twitter deve ser construída aqui, porém não carreguem muitos dados agora. Vamos deixar para carregar mais dados quando o Processamento estiver implementado na próxima sprint.

DICA: Carregue poucos dados. Sugestão: apenas 30 minutos de dados

- 5) **Desafio XPTO:** Capturar Tweets sobre o Presidente do Brasil e armazená-los na RAW Zone. Persistir os tweets no máximo 100 em cada arquivo JSON e no nome do arquivo incluir a data com hora e minuto dos tweet mais antigo no arquivo.
Dicas Glue: Lembrando que para isso Não será preciso uso de Spark apenas com o Python é possível. Apenas o Glue Python pode resolver. Dicas:
- Criar Role com Permissões de GLUE, S3 e CloudWatch
- Incluir Libs extra para adicionar: tweepy
Dicas EMR: Utilize a mesma configuração EMR já criada anteriormente e quando for executar os códigos use apenas uma EMR para tudo.
- 6) **Desafio XPTO:** Carregar os dados que chegam na RAW Zone para a REF Zone de maneira contínua (como streaming), unificando os dados num Bucket S3 persistindo no formato parquet. Particionar os dados por Ano/Mês/Dia conforme a criação do Twitter. Incluir 2 colunas novas:
- Sentimento: indicando Positivo quando encontrar algum símbolo como :D ou :) ou :) etc; Indicando Negativo quando encontrar algum símbolo como :(ou :[ou :{ etc. Indicando Neutro, quando o tweet não tiver nenhum dos símbolos analisados. Se um tweet tiver vários símbolos apenas o primeiro encontrado deve ser utilizado
- Símbolo: Nesta coluna você deve adicionar o símbolo encontrado no tweet. Se um tweet tiver vários símbolos apenas o primeiro encontrado deve ser utilizado
- 7) **Desafio XPTO:** Habilitar o Amazon Athena para fazer consultas SQL na camada REF Zone.



Referências

- [1] <https://spark.apache.org/docs/latest/streaming-programming-guide.html>
- [2] <https://spark.apache.org/docs/latest/submitting-applications.html>
- [3] <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>
- [4] <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#output-sinks>
- [5] <http://bit.ly/2mYSjyO>
- [6] <https://docs.databricks.com/applications/machine-learning/mllib/binary-classification-mllib-pipelines.html>
- [7] <https://help.twitter.com/pt/rules-and-policies/twitter-api>
- [8] <https://pt.wikipedia.org/wiki/Twitter>