

Alunos: Rafael Leão Teixeira de Magalhães - 190019158

Abner Filipe Cunha Ribeiro - 190041871

João Pedro Moura Oliveira – 190030879

Spark Streaming e Apache Kafka

1 - Introdução

O projeto tem como principal função o estudo mais a fundo das arquiteturas de clusters de processamento de *streaming* e programação de aplicações para consumo e tratamento de eventos, em tempo real. Para tanto, o projeto foi dividido em duas partes: a **primeira** utilizando apenas o Spark Streaming contabilizando palavras de entrada via *socket*, atuando como streaming e processamento dos dados. Enquanto na **segunda** ocorreu a composição do Spark Streaming, para o processamento das palavras, em conjunto com o Apache Kafka para o *streaming* dos dados.

O Apache Spark é um mecanismo de análise unificado para processamento de dados em grande escala com módulos integrados para SQL, *streaming*, *machine learning* e processamento de gráficos. O Spark pode ser executado no Apache Hadoop, Apache Mesos, Kubernetes, por conta própria, na nuvem e em diversas outras fontes de dados. Por sua vez, o Apache Kafka é uma plataforma distribuída de transmissão de dados que é capaz de publicar, subscrever, armazenar e processar fluxos de registro em tempo real. Essa plataforma foi desenvolvida para processar fluxos de dados provenientes de diversas fontes e entregá-los a vários clientes.

2 – Metodologia

Como metodologia de desenvolvimento, o grupo procurou se reunir ao menos três vezes por semana para discussão e nivelamento dos problemas e da evolução da solução do trabalho desenvolvida. Nos encontros foram apresentados a situação das etapas do projeto até o momento e, em conjunto, ocorreu a tentativa de resolução dos possíveis impedimentos que cada um estava tendo. Assim como, em conjunto, desenvolvemos parte do projeto para que todos conseguissem entender melhor o projeto como um todo.

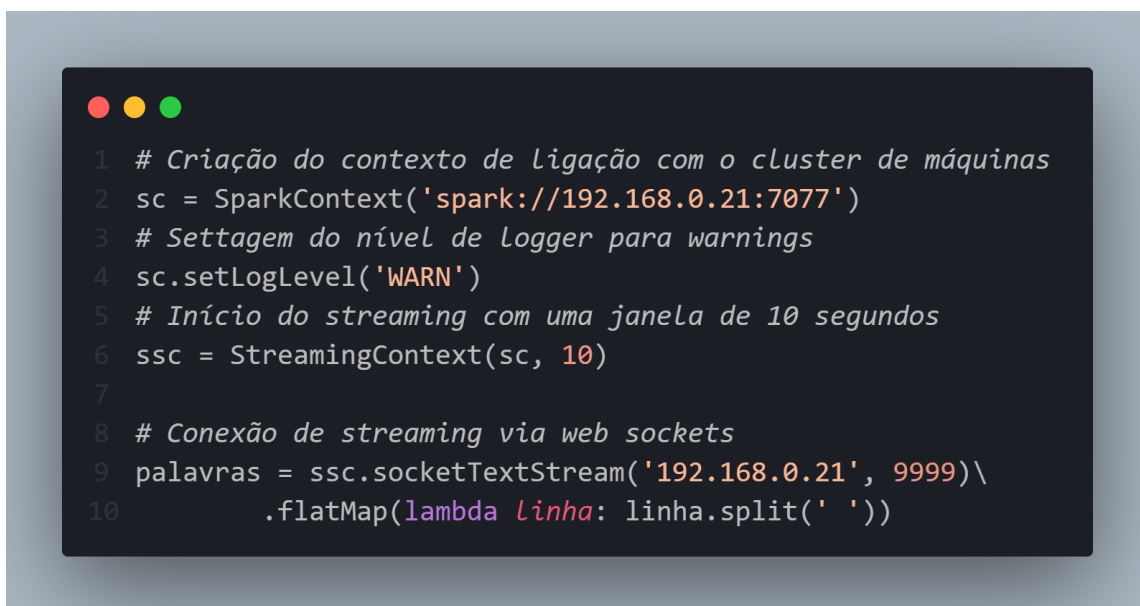
3 – Descrição da Solução

Para o desenvolvimento da solução quatro principais ferramentas/*frameworks* foram utilizados, sendo eles: o Spark Streaming para processamento dos dados, o Apache Kafka para o *streaming* das informações, o netcat/telnet para comunicação via *sockets* utilizada na primeira etapa do projeto, o docker em conjunto com o docker-compose para a orquestração dos containers, e para o frontend foram utilizados o NodeJS e também o *framework* ExpressJS para roteamento e gerência de API e o React para o *frontend*. Vale a pena citar que, o docker só foi utilizado para subir várias instâncias de um *worker* Spark, bem como, para virtualizar o Apache Kafka em conjunto com o Zookeeper.

3.1 – Solução de *Sockets* com o Spark Streaming

Para a solução da primeira parte do projeto, Spark Streaming em conjunto com *sockets*, foi desenvolvido um código utilizando o PySpark [1] e seu conceito de DStream ou *Discretized Stream*. As DStream são a abstração básica do Spark para *streamings* de informações, sendo uma API completa que já possui as implementações dos conceitos de *map* e *window*, importantíssimos para a implementação final. A seguir, na figura 1, pode ser observada essa parte do código que cria a conexão e inicia o *streaming* dos dados.

FIGURA 1: Criação da DStream e início do *streaming*



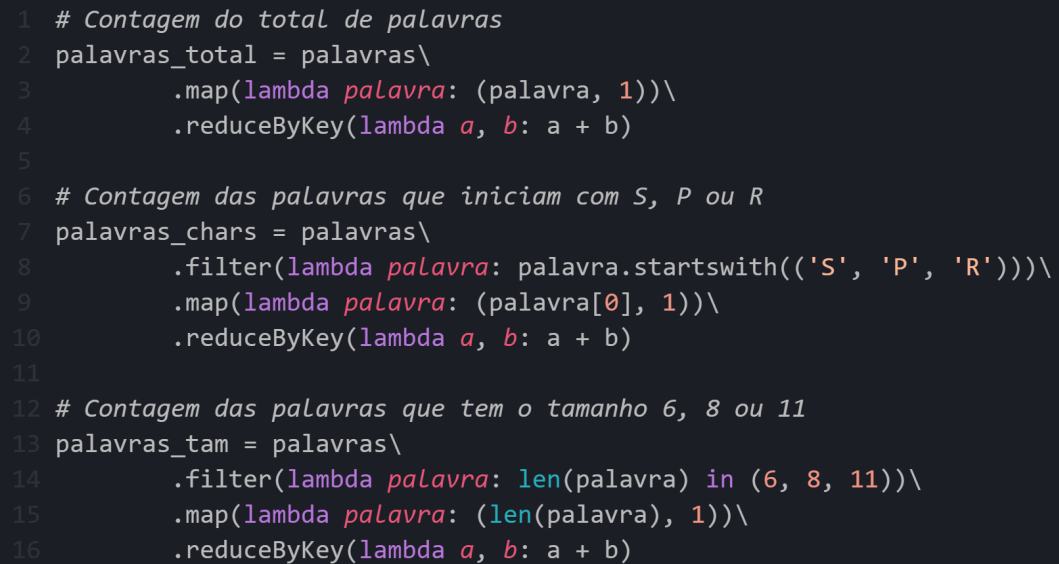
```
1 # Criação do contexto de ligação com o cluster de máquinas
2 sc = SparkContext('spark://192.168.0.21:7077')
3 # Settagem do nível de logger para warnings
4 sc.setLogLevel('WARN')
5 # Início do streaming com uma janela de 10 segundos
6 ssc = StreamingContext(sc, 10)
7
8 # Conexão de streaming via web sockets
9 palavras = ssc.socketTextStream('192.168.0.21', 9999)\
10     .flatMap(lambda linha: linha.split(' '))
```

Fonte: Autor.

Com o fluxo de informações já encaminhado é possível, portanto, realizar as transformações esperadas nos dados, com os comandos de map e reduce, filters e muitos

outros comandos SQL já disponíveis na API do Spark Streaming. Todas essas transformações, podem ser observadas no trecho de código apresentado na figura 2.

FIGURA 2: Transformações dos dados.

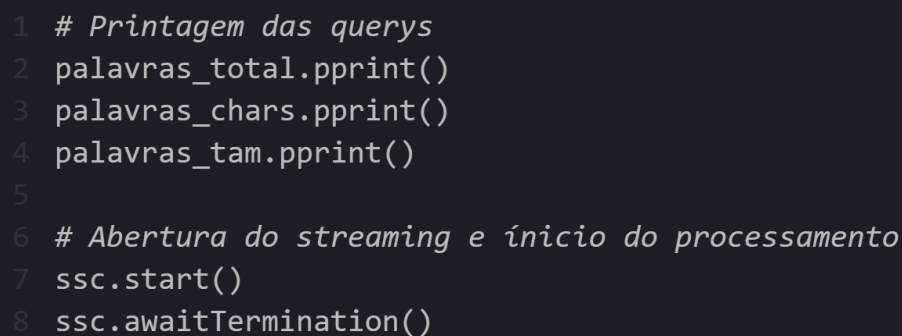


```
1 # Contagem do total de palavras
2 palavras_total = palavras\
3     .map(lambda palavra: (palavra, 1))\
4     .reduceByKey(lambda a, b: a + b)
5
6 # Contagem das palavras que iniciam com S, P ou R
7 palavras_chars = palavras\
8     .filter(lambda palavra: palavra.startswith(('S', 'P', 'R')))\
9     .map(lambda palavra: (palavra[0], 1))\
10    .reduceByKey(lambda a, b: a + b)
11
12 # Contagem das palavras que tem o tamanho 6, 8 ou 11
13 palavras_tam = palavras\
14    .filter(lambda palavra: len(palavra) in (6, 8, 11))\
15    .map(lambda palavra: (len(palavra), 1))\
16    .reduceByKey(lambda a, b: a + b)
```

Fonte: Autor.

Por fim, as últimas etapas necessárias de serem executadas são a impressão no terminal das palavras analisadas e o início real e assíncrono do código e do *streaming*.

FIGURA 3: Impressão e início do *streaming*.



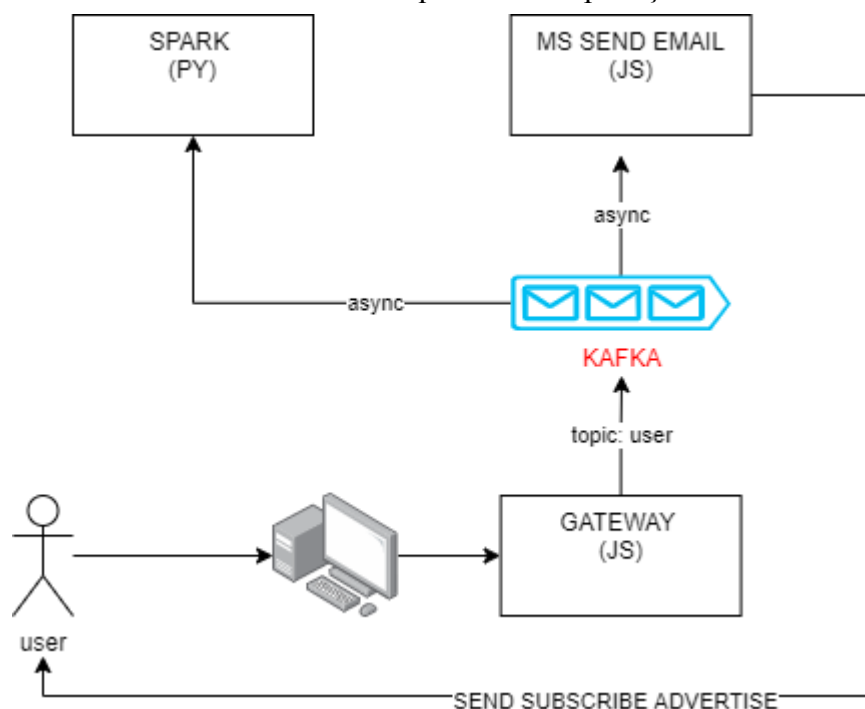
```
1 # Printagem das queries
2 palavras_total.pprint()
3 palavras_chars.pprint()
4 palavras_tam.pprint()
5
6 # Abertura do streaming e início do processamento
7 ssc.start()
8 ssc.awaitTermination()
```

Fonte: Autor.

3.2 – Solução Apache Kafka e Spark Streaming

Dessa maneira, para a solução final do trabalho inicia-se o uso da tecnologia de *streaming* do Apache Kafka [3]. Ao contrário da parte 1, agora as informações são advindas de um sistema complexo de orquestração, que provê um alto *throughput* (entrada/saída) de dados, alta escalabilidade, bem como, uma alta disponibilidade. Esse *framework*, portanto, permite o desenvolvimento de uma aplicação bem mais próxima com a realidade de diversos serviços de software famosos no mercado. Em nossa utilização implementamos um Front-end desenvolvido em React, plataforma em que o usuário irá fazer a entrada de dados para execução de todo nosso fluxo, após isso as informações preenchidas pelo usuário serão enviadas para o gateway que fará a publicação no tópico ‘user’ do kafka, o fluxo completo pode ser conferido no diagrama abaixo.

FIGURA 4: Arquitetura da aplicação



Fonte: Autor.

Também em contradição às DStream (descontinuada em versões mais atuais do Kafka) apresentadas e utilizadas na primeira parte do trabalho, para a segunda etapa foram utilizadas as *Structured Streaming* [2]. Essa API é construída em cima do motor SQL do Spark, provendo uma menor latência, com uma maior entrada e saída de dados, garantia de entrega das mensagens e resiliência a falhas. Porém, para alcançar isso a programação precisou se tornar um pouco mais complexa a fim de garantir todas as vantagens.

Por esse motivo, o código teve que ser alterado utilizando conceitos de classes do python. Nas figuras 5 e 6, podem ser observadas todo o fluxo de inicialização do *streaming* Kafka em conjunto com o Spark.

FIGURA 5: Inicialização do Spark com o Kafka

```
1 def __init__(self, argv):
2     self.topicos = ','.join([argv[i] for i in range(3, len(argv))])
3     self.broker = argv[2]
4     self.timestamp = argv[1]
5
6     # Cria uma sessão que liga ao cluster spark
7     self.spark = SparkSession.builder \
8         .master('spark://192.168.0.21:7077') \
9         .appName('SparkKafkaCluster') \
10        .getOrCreate()
11
12    # Setta o Logger para warnings, inicializa o data frame de streaming
13    # e já le as palavras que serão processadas (JSON)
14    self.spark.sparkContext.setLogLevel('WARN')
15    self.df = self.__inicializa_dataframe()
16    self.palavras = self.df.select(explode(split(
17        from_json(self.df.value.cast('string'), 'resumo STRING').resumo,
18        ' '
19    )), alias('value'), self.df.timestamp)
```

Fonte: Autor.

FIGURA 6: Inicialização do data frame como *streaming*.

```
1 # Inicialização de data frame de streaming do kafka, são settadas
2 # algumas configurações extras também
3 def __inicializa_dataframe(self):
4     return self.spark.readStream \
5         .format('kafka') \
6         .option('kafka.bootstrap.servers', self.broker) \
7         .option('subscribe', self.topicos) \
8         .option('startingOffsets', 'latest') \
9         .option('failOnDataLoss', 'false') \
10        .option('includeTimestamp', 'true') \
11        .load() \
```

Fonte: Autor.

Realizada a conexão e criado o *Data Frame* parte-se então para a filtragem, agrupamento e seleção dos dados que serão recuperados por cada *query* específica, esse processo pode ser visto no pedaço de código representado na figura 7.

FIGURA 7: Filtragem das *queries*.

```
1 # A partir desse ponto, são queries específicas que possuem suas ordenações,  
2 # filtragens, entre outros comandos de bancos adicionados aqui  
3 @property  
4 def filtra_por_char(self):  
5     return self.palavras \  
6         .select(col('timestamp'), col('value').substr(1, 1).alias('value')) \  
7         .where(col('value').rlike('^[SPR]')) \  
8  
9 @property  
10 def filtra_por_quantidade(self):  
11     return self.palavras \  
12         .select(col('timestamp'), length(col('value')).alias('value')) \  
13         .where(col('value').isin([6, 8, 11])) \  

```

Fonte: Autor.

Ao contrário das DStream, para a API de *Structured Streaming* é necessário configurar as janelas de processamentos com *watermark* para evitar a perda de mensagens [4], e só após isso, realizar a criação do *streaming* de saída das informações. Com isso, todas as *queries* foram montadas e pode-se iniciar o processamento de cada uma de forma assíncrona e paralelizada graças ao cluster provido pelo Spark. Toda a codificação pode ser vista na figura 8 e 9 mostradas a seguir.

FIGURA 8: Adição das *windows* nas *queries*.

```
1 # Método que aplica a timestamp com watermark para respostas atrasadas.  
2 # Além disso, também é definido a janela, realizado o agrupamento por  
3 # palavras, sua contagem e ordenação por janela.  
4 def _aplica_timestamp(self, query):  
5     return query \  
6         .withWatermark('timestamp', f'{self.timestamp} seconds') \  
7         .groupBy(  
8             window(col('timestamp'), f'{self.timestamp} seconds', f'{self.timestamp} seconds'),  
9             col('value')) \  
10        .count().sort(desc('window'))  

```

Fonte: Autor.

FIGURA 9: *Streaming e execução das queries.*

```
1 # Método responsável por receber um array de queries e adicionar os
2 # comandos de streaming de saída, bem como formatação JSON
3 def retorna_stream_de_escrita(self, queries):
4     return [self._aplica_timestamp(getattr(self, q)) \
5             .select(to_json(struct('*')).alias('json')) \
6             .selectExpr('json AS value') \
7             .writeStream \
8             # .option("checkpointLocation", "hdfs://192.168.0.21:9000/spark/") \
9             # .format('kafka') \
10            # .option('kafka.bootstrap.servers', self.broker) \
11            # .option('topic', 'testcluster2') \
12            .outputMode('complete') \
13            .format('console') \
14            .option('truncate', 'false') \
15            .start()
16     for q in queries
17 ]
18
19 # As queries vão ser processadas apenas nessa função com o awaitTermination
20 def espera_execucao(self, streams):
21     for s in streams:
22         s.awaitTermination()
```

Fonte: Autor.

4 – Conclusão

Por fim, como resultados é perceptível que o projeto foi concluído com sucesso, visto que, o cluster Spark em conjunto com o Apache Kafka atingiram os objetivos de sistemas distribuídos com alta escalabilidade, baixa latência e com alta disponibilidade. Além disso, é perceptível que a aquisição desses conhecimentos constitui uma base importante para a formação acadêmica dos integrantes.

Apesar do sucesso, diversas limitações foram encontradas, sendo algumas delas que valem ser ressaltadas: não foi possível a criação de clusters com mais de uma máquina física, sendo o mesmo criado apenas com máquinas virtualizadas graças ao docker, além disso, diversos problemas de compatibilidade de versões entre *frameworks* e linguagens foram encontrados e solucionados. Problemas de conexão do Kafka entre computadores em diferentes redes também foram identificados e infelizmente não foram possíveis de serem corrigidos.

E por fim, vale a pena eliciar que a geração de gráficos também não foi feita, já que, a forma como foi desenhado o micro serviço comunicando com o Spark por uma segunda fila acabou comprometendo o desenvolvimento eficaz e funcional do mesmo. Toda a solução pode ser acessada no seguinte endereço virtual: <https://github.com/abner423/poc-kafka-node>.

4.1 – João Pedro Moura Oliveira

Eu gostei bastante do trabalho principalmente pelas tecnologias utilizadas serem bastante atuais e comumente utilizadas por grandes empresas, mostrando o potencial de processamento das mesmas. A única sugestão de melhoria que eu consigo encontrar no momento é em relação ao tempo para desenvolvimento do trabalho. Foi legal, mas caso a especificação tivesse sido apresentada uma ou duas semanas antes, mais funcionalidades poderiam ser desenvolvidas.

Como comentários, acredito que seria interessante ressaltar que essas tecnologias são bem chatas no quesito de compatibilidade e versão das linguagens e *frameworks*, com algumas delas não possuindo funcionalidades (a última versão do Spark com Kafka descontinuou o DStream). Em relação a minha participação, eu estive presente em todas as etapas do desenvolvimento com exceção do frontend. Por fim, acredito que para minha nota de autoavaliação seria 10.

4.2 – Abner Filipe Cunha Ribeiro

Apreendi bastante com o desenvolvimento do projeto, porque foi possível a separação das demandas como um time distribuído, em que cada integrante atuou nas suas demandas, e resolvemos de forma descentralizada, realizando os testes de forma mockada. Porém achei o tempo de execução curto, o que nos levou a diminuirmos o escopo que havíamos planejado inicialmente, ficando de sugestão para as próximas turmas, a apresentação do tema e escopo do projeto desde o início do semestre para um melhor planejamento.

Atuei em todas as etapas do desenvolvimento, desde a entrada dos dados pelo usuário, até o processamento pelo spark, tivemos vários problemas com a implementação do spark, mas parte delas foi resolvida utilizando o padrão structured. Tendo em vista a nossa de implementação do fluxo além do proposto pelo professor, julgo que a nota do grupo deveria ser 10, e tendo em vista minha atuação 10 também.

4.1 – Rafael Leão Teixeira de Magalhães

Achei a temática e as tecnologias utilizadas muito interessantes. Dentre o kafka e o spark eu já tinha algum conhecimento prévio sobre o kafka, mas isso não tornou a experiência de realizar o projeto mais fácil. Uma vez que, como o João citou, tivemos problemas com a compatibilidade entre essas tecnologias.

Atuei em todas as etapas do desenvolvimento do projeto, desde a entrada de dados pelo usuário no Front-end, na criação do gateway e serviço de envio de e-mails, até as etapas finais de processamento de dados pelo spark. Acho importante frisar que a equipe trabalhou muito bem e realizou reuniões bastante importantes e sincronizadas. Acredito que devido à qualidade do trabalho apresentado, e à implementação além do proposto a nota do grupo deveria ser 10, e atribuo à minha nota individual também a nota 10.

5 – Referências

[1] APACHE SPARK. **Spark Streaming. Programming Guide**. Disponível em: <https://spark.apache.org/docs/latest/streaming-programming-guide.html>. Acesso em: 23 abr. 2022.

[2] APACHE SPARK. **Structured Streaming Programming Guide**. Disponível em: <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>. Acesso em: 22 abr. 2022.

[3] APACHE SPARK. **Structured Streaming + Kafka Integration Guide (Kafka broker version 0.10.0 or higher)**. Disponível em: <https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html>. Acesso em: 24 abr. 2022.

[4] APACHE SPARK. **Types of time windows**. Disponível em: <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#types-of-time-windows>. Acesso em: 29 abr. 2022.