Qual o objetivo do comando *cache* em *Spark*?

Segundo Apache (2018a), o comando cache permite que armazenar um conjunto de dados na memória cache do *cluster* como um todo. Esta funcionalidade é útil quando o conjunto de dados é acessado repetidamente, ou ao executar algoritmos iterativos.

O mesmo código implementado em *Spark* é normalmente mais rápido que a implementação equivalente em *MapReduce*. Por quê?

Segundo Dean e Ghemawat (2004), entre as etapas de Map e de *Reduce*, os arquivos intermediários resultantes do processo de mapeamento são persistidos em disco, antes de serem distribuídos para as diversas máquinas na etapa de *Reduce*. Uma vez que o *Spark* por padrão procura realizar todo o processamento em memória, e também pela característica de *lazy evaluation* do RDD, isto é, de realizar o processamento apenas no final, também contribui para melhoria de performance (APACHE, 2018b).
Também é fato que o *MapReduce* perde performance quando executado várias vezes para processar um algoritmo mais complexo, que possua várias etapas, enquanto o *Spark* pode ser até 100 vezes mais rápido.

Qual é a função do *SparkContext* ?

Segundo Apache (2014), a classe *SparkContext* é o ponto de entrada principal para as funcionalidades do *Spark*. A *SparkContext* representa uma conexão com um *cluster Spark*, e pode ser utilizado para criar RDDs e variáveis de *broadcast* neste *cluster*.

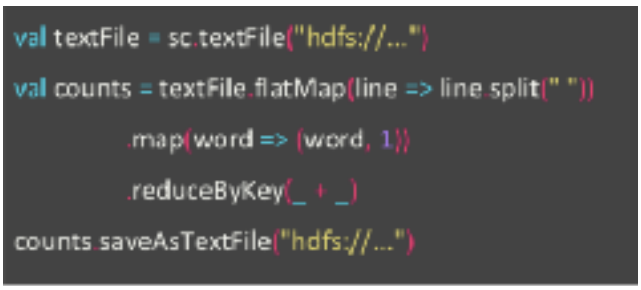
Explique com suas palavras o que é *Resilient Distributed Datasets* (RDD).

O RDD é um conceito desenvolvido por Zahara et al. (2012) que busca oferecer um método mais eficiente de se trabalhar com reutilização de dados intermediários em múltiplas operações computacionais. O RDD é uma estrutura de dados paralela tolerante a falhas que permite ao usuário persistir explicitamente os resultados intermediários em memória, controlar seu particionamento, otimizar a alocação de dados e manipulá-los através de um conjunto de operadores.

***GroupByKey* é menos eficiente que *reduceByKey* em grandes dataset. Por quê?**

O método *reduceByKey* combina todas as chaves antes de realizar o *shuffle*, enquanto o *GroupByKey* realiza primeiramente o *shuffle* para depois organizar os pares de chave-valor. Sendo assim, em datasets grandes, a eficiência do *reduceByKey* tende a ser melhor que o do *GroupByKey* uma vez que envia pela rede apenas o resultado da função de redução, ao contrário do *GroupByKey*, que envia o conjunto de dados como um todo.

Explique o que o código Scala abaixo faz.



O algoritmo obtém a contagem de cada uma das palavras presentes em um documento de texto, e salva o resultado em outro arquivo de texto.

Passo-a-passo:

Obtém um ponteiro para um arquivo de texto em um sistema de arquivos HDFS utilizando a classe SparkContext, e o atribui a uma variável textFile (RDD).
Em seguida há um encadeamento de funções que realiza todo o processo de quebrar o texto por espaços em branco, e então realizar a contagem das palavras. São realizadas as seguintes etapas:

1. A partir da coleção de linhas, obtém uma coleção de todas as palavras do texto utilizado a função *flatMap*, e passando por parâmetro uma função *lambda* que quebra cada linha do arquivo por espaços em branco. (A função *flatMap* transforma um RDD de tamanho N em uma coleção com N coleções e então “achata" todas elas um único RDD com os resultados)

2. Aplica a transformação map, que cria pares de chave-valor, sendo que a palavra representa a chave, e o número 1 representa o valor.

3. Aplica a ação *reduceByKey* que agrega os valores pela soma para cada chave, realiza o *shuffle,* e então retorna um dataset contendo os pares de chave-valor (palavra e contagem) para serem armazenados na variável *counts*.

Obs: as etapas 1 e 2 só são efetivamente executadas quando a ação *reduceByKey* é chamada.

Por fim, o algoritmo salva o dataset contendo os pares de palavras com suas respectivas contagens em um arquivo de texto em um sistema de arquivos HDFS.

REFERÊNCIAS

APACHE. **Quick Start.** 2018a. Disponível em: <https://spark.apache.org/docs/latest/quick-start.html>. Acesso em: 5 maio 2018.

APACHE. **RDD Programming Guide.** 2018b. Disponível em: <https://spark.apache.org/docs/2.3.0/rdd-programming-guide.html>. Acesso em: 5 maio 2018.

APACHE. **Class SparkContext.** 2014. Disponível em: <https://spark.apache.org/docs/1.0.1/api/python/pyspark.context.SparkContext-class.html>. Acesso em: 5 maio 2018.

DEAN, Jeffrey; GHEMAWAT, Sanjay. MapReduce: Simplified Data Processing on Large Clusters. **OSDI'04**: Sixth Symposium on Operating System Design and Implementation, San Francisco, Ca, v. 1, n. 1, p.1-1, dez. 2004. Disponível em: <https://static.googleusercontent.com/media/research.google.com/pt-BR//archive/mapreduce-osdi04.pdf>. Acesso em: 5 maio 2018.

ZAHARA, Matei et al. Resilient distributed datasets. In: PROCEEDINGS OF THE 9TH USENIX CONFERENCE ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION, 9., 2012, San Jose, Ca. **Resilient distributed datasets.** A Fault-tolerant Abstraction For In-memory Cluster Computing: Usenix, 2012. Disponível em: <https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf>. Acesso em: 5 maio 2018.