# SEMANTIX TESTE – SOLUÇÕES


## CARLOS HENRIQUE BARROSO SENA SOUSA
**henriquebsena@gamil.com**


## PARTE TEÓRICA

### QUESTÃO 1.

A função cache é utilizada no *Spark* para otimizar alguma aplicação de interesse. Ela é outra forma de *otimizar* o processo, assim como a função *persist*. Contudo a função cache tem como padrão o trabalho com os RDD que atendem a opção *Memory_Only* no *SotrageLevel*.

### QUESTÃO 2.

Depende. Com a utilização do método *Spark*, o processamento de tarefas será realizado em memória RAM. Já com a utilização do método *Mapreduce*, o processamento de tarefas será realizado em Disco (HD). Essa diferença (no uso de memória RAM e Disco) pode garantir a vantagem do método *Spark* em diversos casos, tornando o processamento mais rápido. Contudo, esta vantagem tende a desaparecer quando o processamento envolve um número muito grande de arquivos, assim tornando o método *Mapraduce* mais atrativo. Portanto, vai depender da finalidade do código utilizado.

### QUESTÃO 03.

O *SparkContext* é um objeto do *Spark* que contem as configurações que serão inseridas no nó principal (*Spark Manager*) de modo a organizar os recursos nos outros nós, afim de efetuar o processamento (aplicação) de interesse.

### QUESTÃO 04.

O *RDD* é um arquivo de leitura construído a partir de informações particionadas e espalhadas nos clusters. O *RDD* é o arquivo no qual o *Spak* aplica as tarefas de interesse do usuário. De uma forma leiga, pode-se dizer que o *RDD* é um tipo de data frame ou tabela que o *Spark* constrói para realizar as operações estipuladas pelo usuário.

### QUESTÃO 05.

No *ReduceByKey* os dados em cada partição são organizados de modo a criar a estrutura chave valor (processo de mapeamento), depois ocorre o processo de Shuffle e por fim o resultado é enviado para algum local específico para performar alguma ação interesse (por exemplo o *Reduce*). Já no *GroupByKey*, os dados são organizados diretamente pela chave (processo de *Shuffle*) e em seguida ocorre o processo que aplica a ação de interesse (por exemplo o *Reduce*). Agora no que diz respeito à desvantagem do *GroupByKey*, vale apontar que grandes conjuntos de dados são enviados para as partições do cluster e isso pode acarretar em falta de memória.

**QESTÃO 06.**

***val textFile = sc.textFile("hdfs://...")***

Esta linha apresenta o objeto *textFile* que recebe local com o arquivo de interesse.

***val counts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)***

Esta linha apresenta o objeto *counts* que é construído através de 3 fases. A primeira fase consiste na aplicação de uma função que opera lendo a linha do arquivo e particionando as informações da linha do arquivo, usando como referência de partição " ". A segunda fase consiste na função de mapeamento responsável por criar a estrutura chave valor, no qual temos como chave a palavra da linha lida e como valor a sua frequência que nesse caso é 1. A terceira fase consiste na aplicação da função de agregação responsável por efetuar a redução das informações e criar a estrutura chave e frequência total (número de ocorrências daquela palavra).

***counts.saveAsTextFile("hdfs://...")***

Esta linha consiste no armazenamento das informações obtidas pela construção do objeto *counts*.

## PARTE PRÁTICA

Soluções efetuadas no databricks:

*https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/8644405563918007/1862162071667374/438116328284645/latest.html*