Tutorial 03 – Análise de força de relacionamento usando Spark

Cenário

Seu Gerente: está, naturalmente, emocionado com as descobertas recentes com as quais você os ajudou - basicamente, você economizou muito dinheiro! Eles começam a dar-lhe maiores perguntas e mais recursos (pelo menos é o que esperamos)

Você: esta ansioso para mergulhar em casos de uso mais avançados, mas você sabe que você precisará ainda de mais financiamento pela organização. Você decide ir adiante!

Análise de força de relacionamento usando Spark

Você surge com uma ótima ideia de que seria interessante para a equipe de marketing saber quais produtos são mais comumente comprados juntos. Talvez existam otimizações para serem feitas em campanhas de marketing para posicionar os componentes em conjunto, que gerarão um forte pipeline de *leads*? Talvez eles possam usar dados de correlação de produtos para ajudar as vendas para os produtos menos visualizados? Ou recuperar receitas para o produto que estava no top 10 dos mais vistos, mas não os 10 melhores vendidos do último exercício?

A ferramenta em CDH mais adequada para análise rápida sobre relacionamentos de objetos é ***Apache Spark***. Você pode compor um trabalho ***Spark*** para fazer este trabalho e dar-lhe uma visão sobre os relacionamentos do produto.

Digite o comando abaixo para executar o *shell* do *spark*:

```
spark-shell --master yarn-client
```

Um pouco sobre o Spark

Se você estiver familiarizado com o *MapReduce*, você notará que este exemplo da *Spark* usa conceitos muito parecidos de operações de "map" e "reduce" (as operações 'join' e 'groupBy' são apenas variações especiais de 'reduce'). A principal vantagem, porém, de usar o *Spark* é que o código é mais conciso e os resultados intermédios podem ser armazenados na memória - permitindo-nos fazer sequências complexas e iterativas muito mais rápidas.

Usar *MapReduce* ainda pode ser uma boa opção para trabalhos em lote que usam muito mais dados do que cabe na memória do *cluster* (por exemplo *petabytes* de dados). Estamos usando *Spark-on-YARN*, o que significa que *MapReduce* e *Spark* (como muitos componentes da CDH) compartilham o mesmo gerenciador de recursos, facilitando o gerenciamento de compartilhamento de recursos entre muitos usuários.

Nota: se deixar parado por algum tempo, o ***prompt scala>*** pode ficar coberto de mensagens de log do cluster. Basta pressionar enter para atualizar o prompt.

Uma vez que o **scala>** apareça, cole o seguinte código:

```
// Primeiro vamos importar as classes que precisamos
```

```
import org.apache.hadoop.mapreduce.Job
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat
import org.apache.avro.generic.GenericRecord
import parquet.hadoop.ParquetInputFormat
import parquet.avro.AvroReadSupport
import org.apache.spark.rdd.RDD
```

```
// Então, criamos RDD's para 2 dos arquivos que importamos do MySQL com o Sqoop
// Os RDDs são estruturas de dados da Spark para trabalhar com conjuntos de dados distribuídos
```

```
def rddFromParquetHdfsFile(path: String): RDD[GenericRecord] = {
   val job = new Job()
   FileInputFormat.setInputPaths(job, path)
   ParquetInputFormat.setReadSupportClass(job,
      classOf[AvroReadSupport[GenericRecord]])
   return sc.newAPIHadoopRDD(job.getConfiguration,
      classOf[ParquetInputFormat[GenericRecord]],
      classOf[Void],
      classOf[GenericRecord]).map(x => x._2)
}

val warehouse = "hdfs://quickstart/user/hive/warehouse/"
val order_items = rddFromParquetHdfsFile(warehouse + "order_items");
val products = rddFromParquetHdfsFile(warehouse + "products");
```

```
// Em seguida, extraímos os campos de pedidos e produtos sobre os quais nos preocupamos
// e obtenha uma lista de cada produto, seu nome e quantidade, agrupados por ordem
```

```
val orders = order_items.map { x => (
   x.get("order_item_product_id"),
   (x.get("order_item_order_id"), x.get("order_item_quantity")))
}.join(
  products.map { x => (
   x.get("product_id"),
   (x.get("product_name")))
  }
).map(x => (
   scala.Int.unbox(x._2._1._1), // order_id
   (
      scala.Int.unbox(x._2._1._2), // quantity
      x._2._2.toString // product_name
   )
)).groupByKey()
```

```
// Finalmente, contamos quantas vezes cada combinação de produtos aparecem
// juntas em uma ordem, então nós as classificamos e pegamos os 10 mais comuns
```

```
val cooccurrences = orders.map(order =>
  (
    order._1,
    order._2.toList.combinations(2).map(order_pair =>
      (
        if (order_pair(0)._2 < order_pair(1)._2)
          (order_pair(0)._2, order_pair(1)._2)
        else
          (order_pair(1)._2, order_pair(0)._2),
        order_pair(0)._1 * order_pair(1)._1
      )
    )
  )
)
val combos = cooccurrences.flatMap(x => x._2).reduceByKey((a, b) => a + b)
val mostCommon = combos.map(x => (x._2, x._1)).sortByKey(false).take(10)
```

```
// Vamos imprimir os resultados, 1 por linha, e sair do Spark shell
```

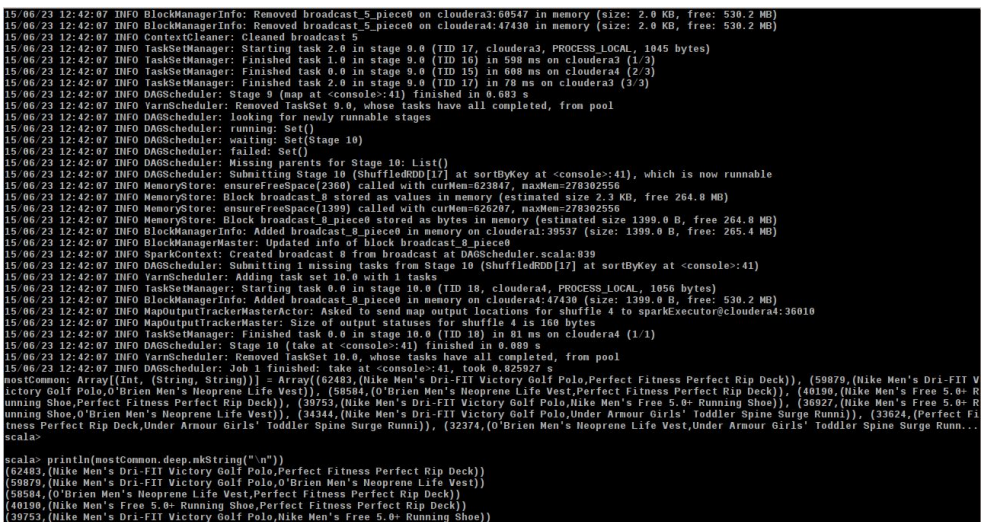
```
println(mostCommon.deep.mkString("\n"))

exit
```

Para entender melhor esse *script*, pode-se ler os comentários que visam explicar o que cada bloco faz e o processo básico em que estamos passando.

Quando fazemos um "map", especificamos uma função que levará cada registro e exibirá um registro modificado. Isso é útil quando precisamos apenas de alguns campos de cada registro ou quando precisamos do registro para usar um campo diferente como a chave: simplesmente invocamos o mapa com uma função que leva todo o registro e retorna um novo registro com o campos e a chave que queremos.

As operações de "reduce" - como 'join' e 'groupBy' - organizarão esses registros por suas chaves para que possamos agrupar registros similares e depois processá-los como um grupo. Por exemplo, agrupamos todos os itens comprados pelo qual a ordem específica foi - permitindo-nos determinar todas as combinações de produtos que faziam parte da mesma ordem.

Você deve ver um resultado semelhante ao seguinte:



CONCLUSÃO

Se não fosse pela *Spark*, fazer análise de ocorrência simultânea como esta seria uma tarefa extremamente árdua e demorada. No entanto, usando Spark e algumas linhas de *scala*, você conseguiu produzir uma lista dos itens mais comprados em conjunto em muito pouco tempo.