

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:

```
15/06/23 12:42:07 INFO BlockManagerInfo: Removed broadcast_5_piece0 on cloudera3:68547 in memory (size: 2.0 KB, free: 530.2 MB)
15/06/23 12:42:07 INFO BlockManagerInfo: Removed broadcast_5_piece0 on cloudera4:47430 in memory (size: 2.0 KB, free: 530.2 MB)
15/06/23 12:42:07 INFO ContextCleaner: Cleaned broadcast_5
15/06/23 12:42:07 INFO TaskSetManager: Starting task 2.0 in stage 9.0 (TID 17, cloudera3, PROCESS_LOCAL, 1045 bytes)
15/06/23 12:42:07 INFO TaskSetManager: Finished task 1.0 in stage 9.0 (TID 16) in 598 ms on cloudera3 (1/3)
15/06/23 12:42:07 INFO TaskSetManager: Finished task 0.0 in stage 9.0 (TID 15) in 580 ms on cloudera4 (2/3)
15/06/23 12:42:07 INFO TaskSetManager: Finished task 2.0 in stage 9.0 (TID 17) in 78 ms on cloudera3 (3/3)
15/06/23 12:42:07 INFO DAGScheduler: Stage 9 (map at <console>:41) finished in 0.683 s
15/06/23 12:42:07 INFO VarScheduler: Removed TaskSet 9.0, whose tasks have all completed, from pool
15/06/23 12:42:07 INFO DAGScheduler: Looking for newly runnable stages
15/06/23 12:42:07 INFO DAGScheduler: running: Set()
15/06/23 12:42:07 INFO DAGScheduler: waiting: Set(Stage 10)
15/06/23 12:42:07 INFO DAGScheduler: failed: Set()
15/06/23 12:42:07 INFO DAGScheduler: Missing parents for Stage 10: List()
15/06/23 12:42:07 INFO DAGScheduler: Submitting Stage 10 (ShuffledRDD[17] at sortByKey at <console>:41), which is now runnable
15/06/23 12:42:07 INFO MemoryStore: ensureFreeSpace(2360) called with curMem=623047, maxMem=278302556
15/06/23 12:42:07 INFO BlockManagerInfo: Block broadcast_8_piece0 stored as values in memory (estimated size 2.3 KB, free 264.8 MB)
15/06/23 12:42:07 INFO MemoryStore: ensureFreeSpace(1399) called with curMem=626207, maxMem=278302556
15/06/23 12:42:07 INFO MemoryStore: Block broadcast_8_piece0 stored as bytes in memory (estimated size 1399.0 B, free 264.8 MB)
15/06/23 12:42:07 INFO BlockManagerInfo: Added broadcast_8_piece0 in memory on cloudera1:35537 (size: 1399.0 B, free: 265.4 MB)
15/06/23 12:42:07 INFO BlockManagerMaster: Updated info of block broadcast_8_piece0
15/06/23 12:42:07 INFO SparkContext: Created broadcast 8 from broadcast at DAGScheduler.scala:839
15/06/23 12:42:07 INFO DAGScheduler: Submitting 1 missing tasks from Stage 10 (ShuffledRDD[17] at sortByKey at <console>:41)
15/06/23 12:42:07 INFO VarScheduler: Adding task set 10.0 with 1 tasks
15/06/23 12:42:07 INFO TaskSetManager: Starting task 0.0 in stage 10.0 (TID 18, cloudera4, PROCESS_LOCAL, 1056 bytes)
15/06/23 12:42:07 INFO BlockManagerInfo: Added broadcast_8_piece0 in memory on cloudera4:47430 (size: 1399.0 B, free: 530.2 MB)
15/06/23 12:42:07 INFO MapOutputTrackerMasterActor: Asked to send map output locations for shuffle 4 to sparkExecutor@cloudera4:36010
15/06/23 12:42:07 INFO MapOutputTrackerMasterActor: Size of output statuses for shuffle 4 is 180 bytes
15/06/23 12:42:07 INFO TaskSetManager: Finished task 0.0 in stage 10.0 (TID 18) in 81 ms on cloudera4 (1/1)
15/06/23 12:42:07 INFO DAGScheduler: Stage 10 (take at <console>:41) finished in 0.489 s
15/06/23 12:42:07 INFO VarScheduler: Removed TaskSet 10.0, whose tasks have all completed, from pool
15/06/23 12:42:07 INFO DAGScheduler: Job 1 finished: take at <console>:41, took 0.825927 s
mostCommon: Array[(Int, (String, String))] = Array((62483,(Nike Men's Dri-FIT Victory Golf Polo,Perfect Fitness Perfect Rip Deck)), (59879,(Nike Men's Dri-FIT V
ictory Golf Polo,O'Brien Men's Neoprene Life Vest)), (58584,(O'Brien Men's Neoprene Life Vest,Perfect Fitness Perfect Rip Deck)), (48189,(Nike Men's Free 5.0+ R
unning Shoe,Perfect Fitness Perfect Rip Deck)), (39753,(Nike Men's Dri-FIT Victory Golf Polo,Nike Men's Free 5.0+ Running Shoe)), (36927,(Nike Men's Free 5.0+ R
unning Shoe,O'Brien Men's Neoprene Life Vest)), (34344,(Nike Men's Dri-FIT Victory Golf Polo,Under Armour Girls' Toddler Spine Surge Runni)), (33624,(Perfect Fi
tness Perfect Rip Deck,Under Armour Girls' Toddler Spine Surge Runni)), (32374,(O'Brien Men's Neoprene Life Vest,Under Armour Girls' Toddler Spine Surge Runn...
scala>
scala> println(mostCommon.deep.mkString("\n"))
(62483,(Nike Men's Dri-FIT Victory Golf Polo,Perfect Fitness Perfect Rip Deck))
(59879,(Nike Men's Dri-FIT Victory Golf Polo,O'Brien Men's Neoprene Life Vest))
(58584,(O'Brien Men's Neoprene Life Vest,Perfect Fitness Perfect Rip Deck))
(48189,(Nike Men's Free 5.0+ Running Shoe,Perfect Fitness Perfect Rip Deck))
(39753,(Nike Men's Dri-FIT Victory Golf Polo,Nike Men's Free 5.0+ Running Shoe))
```

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.