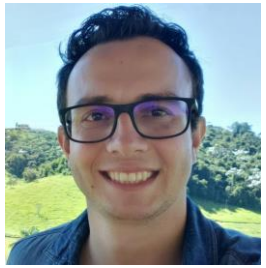
Spark – Big Data Processing

Aula 9

Semantix®

All about data

Quem sou eu?



Rodrigo Augusto Rebouças

Engenheiro de dados da Semantix
Instrutor do Semantix Academy

Contatos
rodrigo.augusto@semantix.com.br
linkedin.com/in/rodrigo-reboucas

RODRIGO.A.REBOUCAS

Semantix®
All about data

Variáveis Compartilhadas

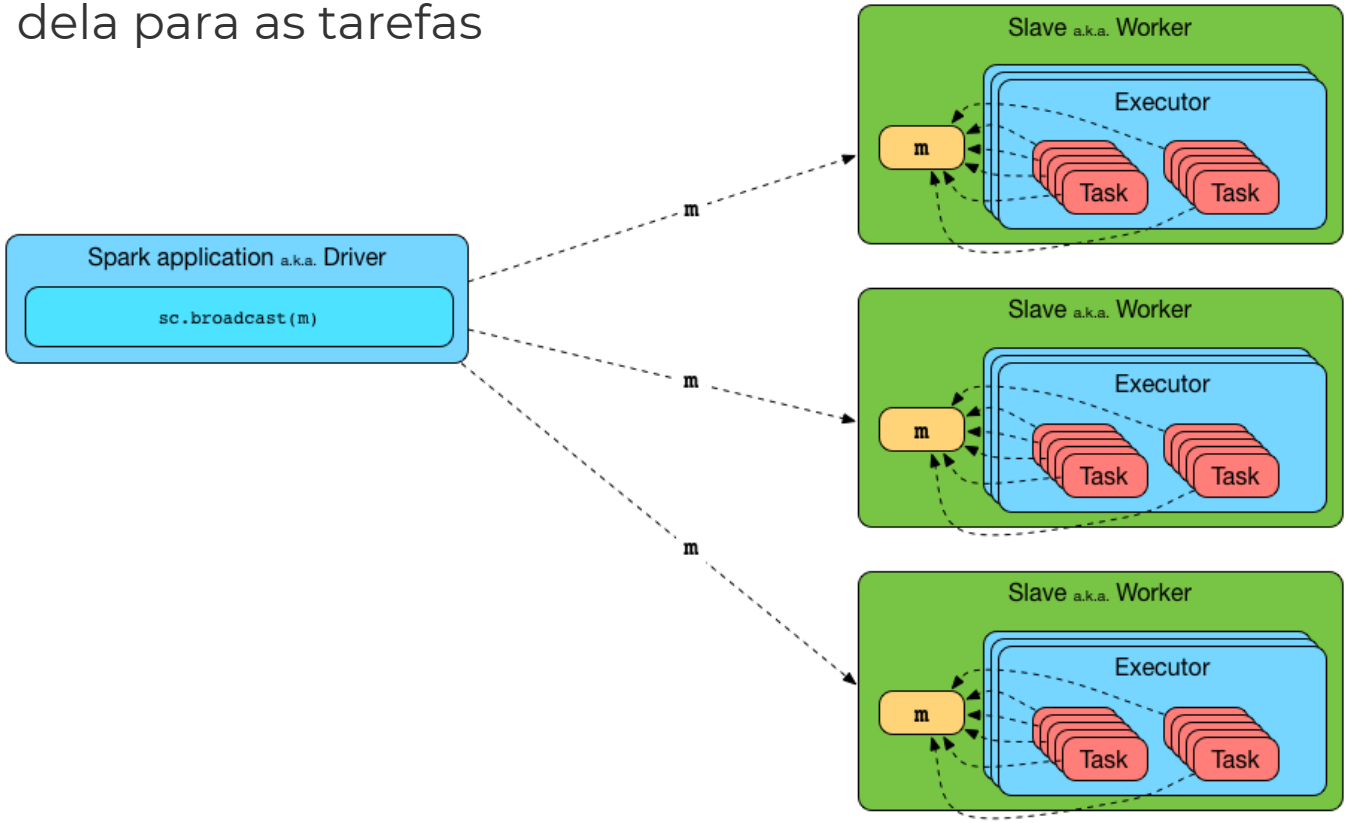
Semantix®
All about data

Variáveis Compartilhadas

- Quando uma função é passada para o Spark, a operação é executada em um nó de cluster remoto
  - Trabalha em cópias separadas de todas as variáveis usadas na função
  - As variáveis são copiadas para cada máquina e nenhuma atualização nas variáveis na máquina remota é propagada de volta ao programa do driver
  - A leitura e gravação entre tarefas é ineficiente
- O Spark fornece dois tipos limitados de variáveis compartilhadas
  - Broadcast
  - Accumulators

Broadcast

- Para cada máquina no cluster terá uma variável somente para leitura em cache
  - Não é necessário enviar uma cópia dela para as tarefas
- Variáveis de broadcast é útil quando
  - Tarefas em vários estágios precisam dos mesmos dados
  - Importância de armazenar em cache os dados na forma desserializada



Broadcast - Métodos

- Id - Identificador único
- Value – Valor
- Unpersist - Exclui assincronamente cópias em cache da variável broadcast nos executores
- Destroy - Destrói todos os dados e metadados relacionados a variável de broadcast
- toString - Representação de string

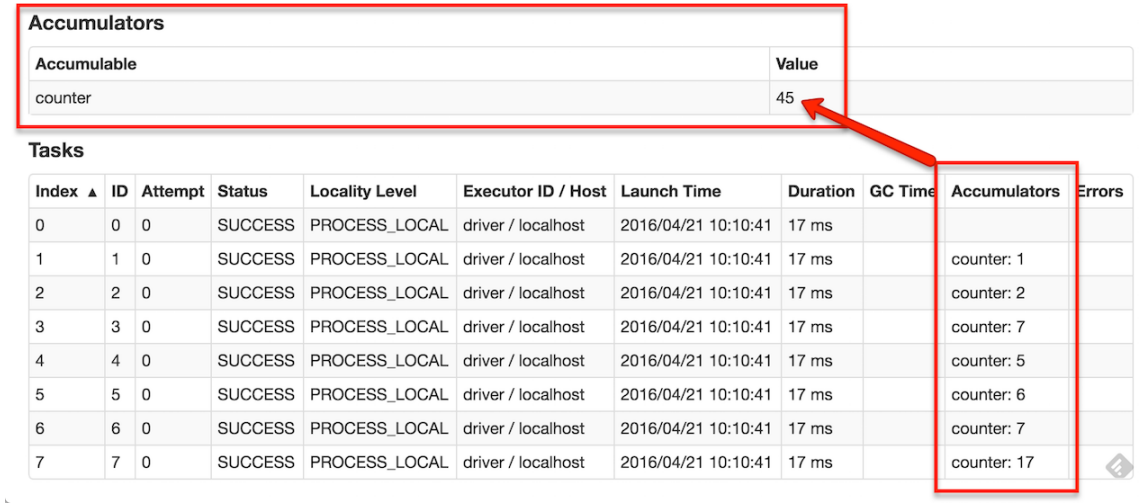
Broadcast - Exemplo

- Sintaxe
- <variavelBroadcast> = sc.broadcast(<valor>)

```
broadcastVar = sc.broadcast( [1, 2, 3])

type(broadcastVar)

pyspark.broadcast.Broadcast

broadcastVar.value

[1, 2, 3]

broadcastVar.destroy
```

Accumulators

- Acumuladores são variáveis que são apenas “adicionadas” a uma operação associativa e comutativa
  - Paralelismo eficiente
  - Podem ser usados para implementar contadores
  - Suporta acumuladores de tipos numéricos, e podem adicionar outros
- O spark exibe o valor para cada acumulador modificado por uma tarefa na tabela “Tasks“
- O rastreamento de acumuladores na interface do usuário pode ser útil para entender o progresso dos estágios em execução



Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

Accumulators

- Criar o acumulador
  - sc. long/doubleAccumulator(valor, “<nomeAcumulador>”) – Scala e view Spark’s UI
  - sc. Accumulator(valor) - Python
- Adicionar tarefas
  - <aculumator.add(Long/Double)

```
scala> val accum = sc.longAccumulator(0, "My Accumulator")
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))
scala> accum.value //10
```

```
python accum = sc. Accumulator(0)
python> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
python> accum.value //10
```

Cache de tabelas

- Armazenar tabela em cache na memória
  - spark.catalog.cacheTable("tableName")
  - dataFrame.cache()
- Remover tabela da memória
  - spark.catalog.uncacheTable("tableName")

```
spark.catalog.cacheTable("src")

broadcast(spark.table("src")).join(spark.table("records"), "key").show()

spark.catalog.uncacheTable("src")
```

UDF

Semantix®
All about data

UDF

- User defined Function para Spark SQL
  - Registrar função como UDF
  - Comando:
    - spark.udf.register(“<nomeUDF>”, <UserDefinedFunction>)

```
scala> val quadrado = ((s: Long) => s * s)
```

```
python> quadrado = (lambda s: s * s)
```

```
spark.udf.register(“fQuad", quadrado)
spark.range(1, 20).registerTempTable("test")
spark.sql("select id, fQuad(id) as id_quad from test")
```

UDF

- User defined Function para DataFrames
  - Comandos:
    - <nomeUDF> = udf(<UserDefinedFunction>)

```
scala>
import org.apache.spark.sql.functions.{col, udf}
scala> val fDfQuad = udf((s: Long) => s * s)
```

```
python>
from pyspark.sql.functions import col,udf
def quadrado (s):
    return s * s
fDfQuad = udf(lambda s: quadrado(s))
```

```
spark.range(1, 20).select(col(“id”), fDfQuad(col("id")))
```

UDF

- Realmente é necessário criar?
  - Otimização
  - Desempenho
  - Documentação
    - https://spark.apache.org/docs/latest/api/sql/

Tunning

Semantix®
All about data

Deploy com alocação dinâmica

- Parâmetros para utilizar os recursos do cluster
  - --master $YARN
    - Executar o log local: $YARN = local
    - Executar no cluster: $YARN = yarn
  - --deploy-mode cluster
  - --conf "spark.dynamic.Allocation.enable=true"
  - --conf "spark.shuffe.service.enable=true"
  - --conf "spark.shuffe.service.port=7337"
  - --conf "spark.dynamic.InitialExecutors=6"
  - --conf "spark.dynamic.maxExecutors=9"
  - --conf "spark.dynamic.minExecutors=3"

Deploy com alocação calculada

- Considerar toda a capacidade da infra/fila
- Parâmetros para utilizar os recursos do cluster
  - --master $YARN
    - Executar o log local: $YARN = local
    - Executar no cluster: $YARN = yarn
  - --deploy-mode cluster
  - --dirver-memory = 8G (recomendável)
  - --executor-memory = int((Memória total-10%) / num-executors)
  - --conf spark.yarn.driver.memoryOverhead = 10% de memoria
  - --conf spark.yarn.executor.memoryOverhead = 10% dos executors
  - --executors-core = 4 ou 5 no máximo, (mais que isso fica pesado)
  - --num-executors = int ((Cores total-10%) / executors-core)

Exercícios

Para as seguintes máquinas:

a) 20 nodes x 8 cores | 16Gb RAM
b) 9 nodes x 20 cores | 128Gb RAM
c) 6 nodes x 10 cores | 32Gb RAM – 3 Jobs em paralelo
d) 10 nodes x 16 cores | 64Gb RAM – 50 % dos recursos já utilizados

Configurar os atributos:

- dirver-memory
- --executor-memory
- --conf spark.yarn.driver.memoryOverhead
- --conf spark.yarn.executor.memoryOverhead
- --executors-core
- --num-executors

Spark Connector

Semantix®
All about data

Conexão com Spark

- Jdbc
  - https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html
- MongoDB
  - https://docs.mongodb.com/spark-connector/current/
- Redis
  - https://github.com/RedisLabs/spark-redis
- Kafka
  - https://spark.apache.org/docs/latest/structured-streaming-kafka-integration.html
- Elastic
  - https://www.elastic.co/guide/en/elasticsearch/hadoop/current/spark.html

Semantix®

All about data

contato@semantix.com.br

www.semantix.com.br