Aqui estão as dicas para otimizar o Spark e economizar recursos, com formatação adequada:

*1. Ajuste o tamanho dos blocos (block size)*

- `spark.sql.files.maxPartitionBytes`: ajuste o tamanho máximo dos blocos de dados lidos do disco.
- Isso pode ajudar a reduzir o número de tarefas e melhorar a eficiência.

Aqui estão algumas dicas para ajustar o tamanho dos blocos (block size) no Spark de acordo com o tamanho do arquivo:

- *Tamanho do arquivo pequeno (< 100 MB)*
  - `spark.sql.files.maxPartitionBytes`: 32 MB a 64 MB
  - `spark.sql.files.openCostInBytes`: 1 MB a 4 MB
- *Tamanho do arquivo médio (100 MB a 1 GB)*
  - `spark.sql.files.maxPartitionBytes`: 64 MB a 128 MB
  - `spark.sql.files.openCostInBytes`: 4 MB a 16 MB
- *Tamanho do arquivo grande (1 GB a 10 GB)*
  - `spark.sql.files.maxPartitionBytes`: 128 MB a 256 MB
  - `spark.sql.files.openCostInBytes`: 16 MB a 64 MB
- *Tamanho do arquivo muito grande (> 10 GB)*
  - `spark.sql.files.maxPartitionBytes`: 256 MB a 512 MB
  - `spark.sql.files.openCostInBytes`: 64 MB a 128 MB

*Regra geral*

- O tamanho dos blocos deve ser entre 1/10 e 1/5 do tamanho do arquivo.
- O custo de abertura de arquivo deve ser entre 1/100 e 1/50 do tamanho do bloco.

*2. Use o cache de dados*

- `spark.cache`: armazene dados frequentemente acessados em memória para reduzir a leitura do disco.

- Use `cache()` ou `persist()` para armazenar dados em memória.

*3. Otimize as junções (joins)*

- `spark.sql.autoBroadcastJoinThreshold`: ajuste o limite para broadcast de tabelas pequenas em junções.

- Use `broadcast()` para forçar o broadcast de tabelas pequenas.

O parâmetro `spark.sql.autoBroadcastJoinThreshold` é usado para determinar se uma tabela deve ser transmitida (broadcast) para todos os nós do cluster durante uma operação de junção (join).

- *O que é broadcast?*

  - Quando uma tabela é transmitida, o Spark a envia para todos os nós do cluster, permitindo que a junção seja realizada localmente em cada nó.

- Como ajustar o `spark.sql.autoBroadcastJoinThreshold`?

  - O valor padrão é 10 MB.

  - Ajuste de acordo com o tamanho das tabelas:

    - Pequena (< 10 MB): provavelmente será transmitida automaticamente.

    - Média (10 MB a 100 MB): aumente para 50 MB ou 100 MB.

    - Grande (> 100 MB): provavelmente não será transmitida automaticamente.

*4. Ajuste o paralelismo*

- `spark.default.parallelism`: ajuste o número de tarefas paralelas para operações como `map()` e `reduce()`.

- `spark.sql.files.openCostInBytes`: ajuste o custo de abertura de arquivos para otimizar o paralelismo.

Aqui estão alguns exemplos de como ajustar os parâmetros:

- *Tamanho dos dados pequeno (< 100 MB)*
  - `spark.default.parallelism`: 2-4
  - `spark.sql.files.openCostInBytes`: 1-4 MB
- *Tamanho dos dados médio (100 MB a 1 GB)*
  - `spark.default.parallelism`: 4-8
  - `spark.sql.files.openCostInBytes`: 4-16 MB
- *Tamanho dos dados grande (1 GB a 10 GB)*
  - `spark.default.parallelism`: 8-16
  - `spark.sql.files.openCostInBytes`: 16-64 MB
- *Tamanho dos dados muito grande (> 10 GB)*
  - `spark.default.parallelism`: 16-32
  - `spark.sql.files.openCostInBytes`: 64-128 MB

*5. Monitore e ajuste o garbage collection (GC)*

- `spark.executor.memory`: ajuste a memória do executor para evitar GC excessivo.
- Monitore o GC com ferramentas como o Spark UI ou o Ganglia.

Aqui estão algumas dicas para ajustar a memória do executor e monitorar o GC:

- *Ajuste da memória do executor*
  - Tamanho dos dados pequeno (< 100 MB): 1-2 GB
  - Tamanho dos dados médio (100 MB a 1 GB): 2-4 GB

- Tamanho dos dados grande (1 GB a 10 GB): 4-8 GB

- Tamanho dos dados muito grande (> 10 GB): 8-16 GB

Sim, há várias configurações de memória RAM do executor que você pode ajustar no Spark:

5.1. spark.executor.memory: define a memória RAM total disponível para cada executor

5.2. spark.executor.memoryOverhead: define a memória adicional para o executor (por exemplo, para o sistema operacional e outros processos)

5.3. spark.memory.fraction: define a fração de memória RAM usada para armazenamento de dados (padrão: 0,6)

5.4. spark.memory.storageFraction: define a fração de memória RAM usada para armazenamento de dados em cache (padrão: 0,5)

5.5. spark.executor.pyspark.memory: define a memória RAM disponível para o Python worker (somente para PySpark)

5.6. spark.executor.pyspark.memoryOverhead: define a memória adicional para o Python worker (somente para PySpark)

Exemplo:

spark.conf.set("spark.executor.memory", "4g") # 4 GB de memória RAM

spark.conf.set("spark.executor.memoryOverhead", "1g") # 1 GB de memória adicional

spark.conf.set("spark.memory.fraction", 0.6) # 60% da memória RAM para armazenamento de dados

spark.conf.set("spark.memory.storageFraction", 0.5) # 50% da memória RAM para armazenamento de dados em cache

Lembre-se de que o ajuste dessas configurações depende do seu ambiente de execução e do tamanho dos dados.

*Díficil de entender?*

- spark.executor.memory: memória RAM total para o executor

- spark.executor.memoryOverhead: memória adicional para o executor

- spark.memory.fraction: fração de memória RAM para armazenamento de dados

- spark.memory.storageFraction: fração de memória RAM para armazenamento de dados em cache

Espero que isso tenha ajudado! 😊

- *Monitoramento do GC*

1. Acesse o Spark UI em `http://<driver-node>:4040`
2. Clique em "Executors"
3. Verifique a coluna "GC Time" para cada executor
4. Se o tempo de GC for alto (> 10%), ajuste a memória do executor

*6. Use o Spark SQL*

- O Spark SQL é mais eficiente do que o RDD API para operações de dados estruturados.

- Use `DataFrame` e `Dataset` para aproveitar as otimizações do Spark SQL.

Lembre-se de monitorar o desempenho do seu aplicativo Spark e ajustar as configurações conforme necessário! 😊