Proposta de Modificação do Algoritmo LRU para o Gerenciamento de Memória no Apache Spark

Maurício Matter Donato[1], **Patrícia Pitthan Barcelos**[1]

[1]Pós-Graduação em Ciência da Computação (PGCC)
Universidade Federal de Santa Maria - UFSM
Santa Maria – RS – Brasil

{mdonato,pitthan}@inf.ufsm.br

Resumo. *Em situações de sobrecarga de memória, o Apache Spark utiliza o algoritmo LRU para gerenciamento do espaço utilizado pelas partições de RDD mantidas em cache. Este trabalho propõe um algoritmo que considera, além da localidade temporal das partições, contemplada pelo LRU, a frequência de acesso às mesmas. Resultados demonstram um desempenho satisfatório, contudo a implementação ainda permite otimizações.*

1. Introdução

O Apache Spark é um *framework* para processamento de grandes quantidades de dados, o qual estende o modelo *MapReduce* do Hadoop. O Spark caracteriza-se pelo processamento em memória principal, através de sua principal abstração, o RDD (*Resilient Distributed Dataset* [Zaharia et al. 2012]). Um RDD consiste em uma coleção imutável de objetos, distribuída entre os nós do *cluster*, onde é possível realizar computações de maneira paralela e distribuída.

O RDD é composto basicamente por uma lista de partições contendo os dados, uma função para computar cada partição e uma lista de dependências (também chamada de *lineage*). RDDs podem ser mantidos em memória principal (*cache*), de modo a possibilitar reusos sem necessidade de recomputação. No entanto, em situações onde não há mais memória disponível, o Spark remove partições mantidas em *cache* de acordo com o algoritmo LRU (*Least Recently Used*). Desta forma, é possível gerar situações onde apenas uma fração do RDD está armazenado em *cache*.

A política de substituição de partições do LRU pressupõe apenas o tempo desde o último acesso, removendo a partição menos recentemente acessada, considerando a localidade temporal. Entretanto, outras métricas podem ser agregadas a este algoritmo, como a frequência de acesso ao RDD da partição ou o custo de recomputá-lo.

Este trabalho propõe um algoritmo para o gerenciamento da memória utilizada pelas partições de RDDs mantidas em *cache*, o qual considera a frequência de acesso às partições e a localidade temporal das mesmas. O trabalho avalia o desempenho do algoritmo proposto através da execução de dois *benchmarks*.

O artigo está organizado da seguinte forma: a Seção 2 apresenta o *framework* Apache Spark e sua abstração, o RDD. A Seção 3 descreve o algoritmo proposto. A Seção 4 apresenta a metodologia de condução dos experimentos, enquanto a Seção 5 descreve os resultados e a discussão acerca dos mesmos. Na Seção 6 são apresentadas as conclusões e os próximos passos.

2. Apache Spark

O Spark é um *framework* para processamento de *big data*, o qual utiliza arquitetura *Master* e *Workers*. O desenvolvimento de aplicações é realizado através de uma API, criando e manipulando de novos RDDs [Chambers and Zaharia 2018]. Novos RDDs são gerados através da aplicação de transformações, enquanto sua computação é realizada com ações.

Quando uma ação é aplicada ao RDD, este é submetido ao *DAGScheduler*, que consiste na camada de escalonamento do Spark e implementa uma abordagem orientada a estágios. O *DAGScheduler* transforma o plano de execução lógico em um plano de execução físico, ou seja, constrói os estágios de execução da aplicação baseando-se na *lineage* e nas dependências do RDD, sendo este processo chamado internamente de *job*. A partir da sequência de estágios gerados, são disparadas *tasks* para serem executadas em nós do *cluster*. A geração do plano de execução físico é realizada toda vez que uma ação for aplicada a um RDD. Assim, uma aplicação Spark pode conter vários *jobs* até sua completa execução.

Cada RDD contém uma lista de partições, responsáveis por armazenar os dados de forma distribuída entre os nós do *cluster*. A fim evitar a recomputação de RDDs, estes podem ser computados e suas partições mantidas em *cache*, permitindo seu reuso. Por padrão, o nível de armazenamento usado pelo Spark é apenas a memória principal. No entanto, a memória possui tamanho limitado e, em situações de sobrecarga, as partições são descartadas de acordo com o algoritmo LRU. O Spark implementa o LRU através de uma lista, a qual armazena as partições mantidas em *cache* de maneira ordenada, cujas partições mais recentemente utilizadas encontram-se nas últimas posições.

3. Algoritmo Proposto

O trabalho propõe um algoritmo que combina tanto a localidade temporal, quanto a frequência de acesso ao RDD. O algoritmo mantém uma lista de partições dos RDDs ordenada pela frequência de acesso de cada RDD utilizado na computação do *job*. Em situações onde há duas ou mais entradas com a mesma frequência, estas são sub-ordenadas de acordo com o algoritmo LRU. Assim, garante-se que os RDDs mais frequentemente utilizados no processamento serão os últimos a serem removidos da memória.

A implementação do algoritmo implica na obtenção da frequência de acesso de cada RDD da aplicação a partir do *DAGScheduler* e na difusão desta informação através de comunicação síncrona entre todos os nós da aplicação. A etapa de difusão se faz necessária uma vez que apenas através do *Master* da aplicação é possível obter informações referentes à frequência de acesso a cada RDD do *Job*. Sendo assim, o sincronismo garante que os dados referentes aos RDDs estejam em todos os nós antes de seu efetivo uso.

Após a transmissão dos dados, conforme as partições dos RDDs utilizadas na computação são carregadas para a memória principal, estas são armazenadas de maneira ordenada pela frequência de acessos ao RDD, obtida através do *DAGScheduler*. Além disso, cada nó do *cluster* possui sua própria lista de partições mantidas em *cache*.

A modificação proposta visa postergar a remoção da memória as partições com alta frequência de acesso. Em contraste, o LRU remove partições com o maior tempo decorrido desde o último acesso, independente da frequência na qual a partição foi acessada em um passado remoto e a possibilidade de reuso em um futuro próximo.

4. Experimentação

A fim de validar o algoritmo proposto, foram conduzidos experimentos utilizando dois *benchmarks* com diferentes formas de acesso à memória: *K-Means* e *PageRank* [Huang et al. 2010]. O *K-Means* consiste em agrupar os dados de entrada em K grupos baseados em suas características. Neste *benchmark*, as partições dos dados são acessadas sequencialmente, sendo este padrão de acesso uma das deficiências do LRU [Jiang and Zhang 2002]. O *PageRank* visa classificar *links* de acordo com suas ligações. A escolha desse *benchmark* se deve à necessidade de avaliar o reuso de partições em futuras computações. Os experimentos visam verificar o impacto, em termos de tempo de execução do algoritmo proposto em relação ao algoritmo LRU implementado pelo Spark.

A experimentação foi realizada na plataforma Grid'5000[1], utilizando um *cluster* com 5 nós configurados da seguinte maneira: 1 Spark *Master*; 2 Spark *workers*; 1 *HDFS Namenode* e 1 *HDFS Datanode*. Cada nó do sistema era composto por um Intel Xeon X3440 @2.53GHz (4 *cores*/CPU) e 16GB de memória RAM, conectados via *ethernet* 1Gbps. O sistema operacional utilizado foi o Debian 8, juntamente com Java JDK 1.8.187, Spark 2.2.0 e Hadoop 2.7.1. Além disso, cada Spark *worker* foi configurado com 1GB e 2GB de memória disponível, de modo a estressar o gerenciamento da mesma. Por fim, os resultados obtidos são a média aritmética de 20 execuções.

5. Resultados

Os resultados obtidos a partir da execução dos *benchmarks* são exibidos na Tabela 1. Conforme a tabela, ambos os experimentos apresentaram tempos de execução satisfatórios, os quais se mostraram similares ao LRU. Tanto no *benchmark K-Means* como no *PageRank*, o tempo médio de execução foi ligeiramente menor com o algoritmo proposto.

Tabela 1. Tempos de Execução dos *benchmarks*

Benchmark	Algoritmo	Mem. Disp.	T. Exec. (seg)	Desvio P. (seg)
K-Means	LRU	1 GB	47.78	0.88
		2 GB	25.48	0.30
	Proposta	1 GB	46.26	2.68
		2 GB	25.21	0.41
PageRank	LRU	1 GB	179.21	2.48
		2 GB	153.91	2.50
	Proposta	1 GB	176.93	3.43
		2 GB	152.78	1.85

Uma análise dos *log* gerados na execução do *K-Means* mostrou que sua execução foi concluída com o processamento de dois *Jobs*, tanto no LRU como no algoritmo proposto. Entretanto, o algoritmo proposto sofre uma sobrecarga, inexistente no LRU, uma vez que há a necessidade de realizar duas comunicações síncronas para atualizar a lista de frequência de acesso aos RDDs, atrasando o início da execução até que a transmissão seja concluída.

[1]Grid'5000 é uma plataforma para experimentos apoiada por um grupo de interesses científicos hospedado por Inria e incluindo CNRS, RENATER e diversas Universidades, bem como outras organizações (mais datalhes em https://www.grid5000.fr).

Nos *logs* de execução do *PageRank* observou-se que sua conclusão se deu em um único *Job*, em ambos os algoritmos. Embora haja uma penalização relacionada à comunicação, o algoritmo proposto obteve uma pequena redução no tempo médio de execução deste *benchmark*, sendo a diferença mais evidente no cenário com 1 GB.

A implementação do algoritmo proposto, em ambos os *benchmarks*, apresenta desempenho satisfatório, porém impõe uma sobrecarga no tempo de comunicação entre nós. A diminuição desta sobrecarga pode reduzir o tempo de execução das aplicações, proporcionando um melhor desempenho. Neste sentido, duas abordagens de otimização devem ser investigadas: comunicação assíncrona e postergação da comunicação. O assincronismo possibilita o início da execução da aplicação mais rapidamente, uma vez que não implica em espera por confirmação. Já a alteração na forma de obtenção das informações referentes à frequência de acesso aos RDDs visa postergar a comunicação entre os nós apenas para situações onde há necessidade de remover partições da memória.

6. Considerações Finais

A busca por soluções mais eficientes para o gerenciamento de memória no Spark tem sido frequentemente pesquisada. Em [Duan et al. 2016] é apresentado o algoritmo *Weight Replacement* (WR), onde o peso do RDD é determinado por três métricas: custo de computação da partição, frequência de acesso e tamanho da partição.

Este trabalho apresentou uma proposta de algoritmo para o gerenciamento da memória no Apache Spark. O algoritmo considera duas métricas para a escolha da partição a ser removida em situação de sobrecarga de memória: frequência de acesso e localidade temporal. Os resultados obtidos demonstraram um desempenho similar ao LRU, embora o algoritmo proposto seja penalizado pela necessidade de comunicação entre os nós do *cluster*. Como trabalhos futuros, pretende-se otimizar a proposta de forma a reduzir a sobrecarga adicionada pela comunicação entre nós.

Referências

Chambers, B. and Zaharia, M. (2018). Spark: The definitive guide: Big data processing made simple.

Duan, M., Li, K., Tang, Z., Xiao, G., and Li, K. (2016). Selection and replacement algorithms for memory performance improvement in spark. *Concurrency and Computation: Practice and Experience*, 28(8):2473–2486.

Huang, S., Huang, J., Dai, J., Xie, T., and Huang, B. (2010). The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th Int. Conference on*, pages 41–51. IEEE.

Jiang, S. and Zhang, X. (2002). Lirs: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association.