UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Maurício Matter Donato

GERENCIAMENTO DINÂMICO DE MEMÓRIA EM APLICAÇÕES COM REUSO DE DADOS NO APACHE SPARK

Santa Maria, RS
2020

Maurício Matter Donato

GERENCIAMENTO DINÂMICO DE MEMÓRIA EM APLICAÇÕES COM REUSO DE DADOS NO APACHE SPARK

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC) da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do título de **Mestre em Ciência da Computação**.

Orientadora: Prof. Dra. Patrícia Pitthan de Araújo Barcelos

Santa Maria, RS

2020

Donato, Maurício Matter

Gerenciamento Dinâmico de Memória em Aplicações com Reuso de Dados no Apache Spark / por Maurício Matter Donato. – 2020.
79 f.: il.; 30 cm.

Orientadora: Patrícia Pitthan de Araújo Barcelos
Dissertação (Mestrado) - Universidade Federal de Santa Maria, Centro de Tecnologia, Pós-Graduação em Ciência da Computação , RS, 2020.

1. Spark. 2. Gerenciamento. 3. Memória. 4. Reuso. 5. Dinâmico. I. Barcelos, Patrícia Pitthan de Araújo. II.Gerenciamento Dinâmico de Memória em Aplicações com Reuso de Dados no Apache Spark.

E-mail: mdonato@inf.ufsm.br

Maurício Matter Donato

GERENCIAMENTO DINÂMICO DE MEMÓRIA EM APLICAÇÕES COM REUSO DE DADOS NO APACHE SPARK

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC) da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do título de **Mestre em Ciência da Computação**.

Aprovado em 25 de Maio de 2020:

Patrícia Pitthan de Araújo Barcelos, Dra. (UFSM)
(Presidente/Orientadora)

João Vicente Ferreira Lima, Dr. (UFSM)

Leandro Krug Wives, Dr. (UFRGS)

Santa Maria, RS

2020

RESUMO

GERENCIAMENTO DINÂMICO DE MEMÓRIA EM APLICAÇÕES COM REUSO DE DADOS NO APACHE SPARK

AUTOR: MAURÍCIO MATTER DONATO
ORIENTADORA: PATRÍCIA PITTHAN DE ARAÚJO BARCELOS

O Apache Spark é um *framework* capaz de processar grandes quantidades de dados em memória, através da sua principal abstração: o *Resilient Distributed Datasets* (RDD). Um RDD consiste em uma coleção imutável de objetos, os quais podem ser operados de maneira paralela e distribuída no*cluster*. Uma vez processados, RDDs podem ser mantidos em *cache*, possibilitando a sua reutilização sem realizar a sua recomputação. Conforme a computação da aplicação é feita, a memória tende a ficar sobrecarregada e, portanto, partições de RDDs devem ser removidas de acordo com o algoritmo *Least Recently Used* (LRU). Este algoritmo é baseado na observação de que partições frequentemente utilizadas em um passado recente tendem a ser acessadas novamente em um futuro próximo. Deste modo, remove-se a partição cujo acesso ocorreu há mais tempo. Entretanto, há situações em que o LRU pode acarretar em uma degradação no desempenho, como é o caso onde há acessos cíclicos à memória e a quantidade de dados manipulados é maior que o espaço disponível. Nessas situações,o LRU sempre irá remover um bloco que será acessado em um futuro próximo. Considerando tal problemática, este trabalho propõe um modelo de Gerenciamento Dinâmico da Memória em Aplicações com Reuso de Dados no Apache Spark. Este modelo busca extrair métricas da aplicação em execução a fim de utilizar estas informações para realizar remoção dos dados em *cache*. O modelo proposto é composto por dois componentes principais, sendo estes (1) um algoritmo de gerenciamento das partições de RDDs armazenadas em memória e (2) um agente de monitoramento responsável por obter informações sobre a execução de aplicações. O modelo de Gerenciamento Dinâmico foi validado através da realização de experimentos utilizando a plataforma Grid'5000 com os *benchmarks PageRank*, *K-Means* e *Logistic Regression*. Os resultados obtidos demonstram que o modelo de Gerênciamento Dinâmico conseguiu realizar um melhor aproveitamento da memória disponível, chegando a reduzir em 23,94% o tempo médio necessário para processar o *benchmark Logistic Regression*, quanto comparado ao LRU. Ademais, o modelo proposto tornou a execução do Spark mais estável, reduzindo a frequência de erros no processamento dos *benchmarks*. Como consequência, houve uma redução de até 34,15% no tempo de execução do *benchmark PageRank*. Portanto, estes resultados permitem concluir que estratégias dinâmicas, como a proposta por este estudo, podem proporcionar um ganho no desempenho do Spark no processamento de aplicações onde existe o reuso de dados.

Palavras-chave: Spark. Gerenciamento. Memória. Reuso. Dinâmico.

ABSTRACT

DYNAMIC MEMORY MANAGEMENT IN APPLICATIONS WITH DATA REUSE ON APACHE SPARK

AUTHOR: MAURÍCIO MATTER DONATO
ADVISOR: PATRÍCIA PITTHAN DE ARAÚJO BARCELOS

The Apache Spark is a framework able to process a massive quantity of data in-memory, through its primary abstraction: Resilient Distributed Datasets (RDD). An RDD consists of an immutable object collection, which can be processed in a parallel and distributed way in the cluster. Once it was processed, an RDD could be stored in the cache, allowing its reuse without recomputing it. While the application's computations are done, the memory tends to be overheaded, and RDD's partitions must be removed according to the Least Recently Used (LRU) algorithm. This algorithm is based on the idea that partitions frequently used in the past will be reaccessed shortly. Thus, the algorithm removes partitions that access occurred a long time ago. However, there are situations that the LRU algorithm could introduce degradation in Spark's performance, which is the case where there is cyclic access in the memory, and the available space is lower than the dataset size. In those situations, the LRU algorithm will always remove the block, which will be accessed soon. Considering the identified issues in the LRU, this work proposes a Dynamic Memory Management in Applications With Data Reuse on Apache Spark. This model aims to extract metrics from the application's execution in order to use that information to realize data removing from the cache. The proposed model is compound by two main components, which are (1) an algorithm to manage the RDD's partitions stored int the memory and (2) a monitor agency responsible for getting information about the application. The Dynamic Management model was validated through experiments using the Grid'5000 platforms with benchmarks PageRank, K-Means, and Logistic Regression. The obtained results demonstrate that the Dynamic Management model was able to improve the utilization of available memory, being able to reduce by 23,94% the necessary execution time to process the benchmark Logistic Regression, when it is compared to LRU. Furthermore, the proposed model became Spark's execution more stable, reducing the error frequency during the processing of benchmarks. As a consequence, there was a reduction by 34,14% in the time spend to process the benchmark PageRank. Therefore, the obtained results allow concluding that dynamic strategies, like the one proposed by this work, can improve the Sparks execution in applications where there is reuse data.

Keywords: Spark. Management. Memory. Reuse. Dynamic.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Program Interface*
ASRW	*Actions Sequence and RDD Weight*
CPU	*Central Processor Unit*
DAG	*Directed Acyclic Graph*
GB	*GigaByte*
GC	*Garbage Collector*
HD	*Hard Drive*
HDFS	*Hadoop Distributed File System*
IP	*Internet Protocol*
JDK	*Java Development Kit*
JSON	*JavaScript Object Notation*
JVM	*Java Virtual Machine*
LCS	*Low Cost Strategy*
LRU	*Least Recently Used*
MB	*MegaByte*
RAM	*Random Access Memory*
REST	*Representational State Transfer*
RDD	*Resilient Distributed Datasets*
RPC	*Remote Procedure Call*
URL	*Uniform Resource Locator*

SUMÁRIO

1 INTRODUÇÃO

Os avanços nas tecnologias de informação permitiram o armazenamento de grandes e variados conjuntos de dados. Com o advento da *internet*, interações *online* estão cada vez mais presentes no cotidiano, possibilitando a comunicação entre pessoas e empresas de maneira fácil e descomplicada. Como consequência dessa interatividade, a quantidade de dados tem crescido a uma taxa significativa durante os últimos anos (GOLDSCHMIDT; BEZERRA; PASSOS, 2015).

Os dados, os quais podem ir desde manifestações em redes sociais até movimentações financeiras, são gerados e disponibilizados em diferentes tamanhos e formatos. De acordo com (MCAFEE et al., 2012), através desses dados, gerentes podem medir e entender seus negócios e, consequentemente, traduzir esse conhecimento em decisões melhores.

Porém, a realização das etapas necessárias para extrair conhecimento a partir dos dados implica em altos custos econômicos e computacionais, uma vez que o armazenamento e o processamento dos mesmos não são tarefas triviais. Para (OUSSOUS et al., 2018), essas dificuldades afetam a captura, o armazenamento, a pesquisa, o compartilhamento, a análise, a gerência e a visualização desses dados. Além disso, a segurança e a privacidade são problemas em aplicações orientadas a dados.

O processamento desses dados requer, por muitas vezes, ferramentas capazes de recorrer ao processamento paralelo e distribuído entre um conjunto de máquinas (*cluster*), além de suportar altas variações no volume de dados utilizado. *Frameworks* baseados no paradigma *MapReduce*, como o Apache Hadoop, têm sido amplamente utilizados para o processamento de grandes volumes de dados. De forma geral, esses *frameworks* oferecem operações de processamento de alto nível e abstrações para acesso aos recursos do *cluster* com o objetivo de facilitar o desenvolvimento de aplicações pelos usuários.

Entretanto, os *frameworks* baseados no paradigma *MapReduce* falham em oferecer abstrações para acesso à memória distribuída tornando-os ineficientes no processamento de algoritmos de reutilização, como aqueles utilizados em Mineração de Dados e Aprendizado de Máquina, (ZAHARIA et al., 2012). Nesse sentido, o Apache Spark[1] surge como um *framework* capaz de processar grandes quantidades de dados de maneira paralela e distribuída estendendo o modelo *MapReduce*, já consolidado pelo Apache Hadoop, de modo a facilitar o desenvolvi-

[1] Disponível em: https://spark.apache.org/

mento de aplicações com reutilização de dados.

O Spark foi projetado para implementar um mecanismo de execução multiestágio em memória principal juntamente com sua principal abstração, o *Resilient Distributed Datasets* ou simplesmente RDD (ZAHARIA et al., 2012)). Dessa forma, o Spark consegue alcançar um desempenho superior quando comparado ao mecanismo baseado em disco utilizado pelo Hadoop. Este desempenho é obtido através da capacidade de processar diversas computações em memória, dispensando a escrita de dados intermediários em disco, diferentemente do Hadoop. Para tanto, o RDD visa encapsular a manipulação dos dados que serão processados pela aplicação, distribuindo-as através dos nodos do *cluster* e assim permitindo a sua execução em memória.

Um RDD consiste em uma coleção imutável de objetos, os quais podem ser operados e processados de forma paralela e distribuída entre os nodos do *cluster*. Uma vez realizado o processamento de um determinado RDD, este pode ser mantido em *cache* para que seja possível reutilizá-lo em futuras computações sem necessidade de efetuar o seu reprocessamento, tornando a execução da aplicação mais eficiente.

Conforme novos RDDs são armazenados e processados, a memória disponível tende a ficar esgotada e, portanto, políticas de gerenciamento de memória devem ser utilizadas. Assim, em situações de sobrecarga no uso do espaço disponível, o Spark remove partições mantidas em *cache* de acordo com o algoritmo LRU (*Least Recently Used*) (LUU, 2018), o qual remove partições cujo acesso ocorreu há mais tempo, explorando apenas a localidade temporal do acesso. Desta forma, é possível gerar situações onde apenas uma fração do RDD permanece armazenado em *cache*.

Entretanto, cabe ao desenvolvedor da aplicação Spark identificar e selecionar os RDDs que deverão ser mantidos em *cache*. De acordo com (ZECEVIC; BONACI, 2016), é importante manter os dados em *cache* para algoritmos iterativos, uma vez que esses tendem a reutilizar os dados. Por padrão, os dados são armazenados em memória principal, uma vez que esta possui uma menor latência para acesso aos dados quando comparado ao armazenamento estável. Entretanto, é possível utilizar o armazenamento estável ou uma combinação deste juntamente com a memória principal.

1.1 JUSTIFICATIVA

Através do RDD, o Spark permite a realização de diversas computações em memória principal a fim de evitar a escrita de resultados intermediários no armazenamento estável.

Ainda, RDDs podem ser mantidos em *cache*, possibilitando a sua reutilização na aplicação sem realizar a sua recomputação. Conforme realiza-se a computação dos RDDs da aplicação, a memória tende a ficar sobrecarregada e, portanto, partições devem ser removidas de acordo com o algoritmo LRU.

O LRU é um algoritmo clássico para o gerenciamento de memória, sendo empregado nos mais variados sistemas (ROBINSON; DEVARAKONDA, 1990). Este algoritmo é baseado na observação de que partições frequentemente utilizadas em um passado recente tendem a ser acessadas novamente em um futuro próximo. Deste modo, a partição removida da memória é aquela cujo acesso ocorreu há mais tempo, uma vez que esta apresenta a menor chance de ser acessada novamente em um futuro próximo.

Embora o LRU seja de fácil implementação (KIM et al., 2000) e apresente bons resultados, há situações onde existe degradação no desempenho. Um exemplo do comportamento indesejado desse algoritmo ocorre quando há acessos cíclicos à memória, como em um *loop*, em que a quantidade de dados manipulados é maior que o espaço disponível em *cache*. Nessas situações,o LRU sempre irá remover um bloco que será acessado em um futuro próximo, uma vez que estes são os blocos acessados há mais tempo (JIANG; ZHANG, 2002). Deste modo, considera-se apenas o tempo desde o último acesso ao bloco de memória, ignorando a sua necessidade em um futuro bem próximo.

Dentre as classes de problemas abordadas pelo Spark, um dos grandes focos é o processamento de aplicações onde há reutilização de dados em iterações futuras. Especificamente nesta classe de aplicações, o comportamento do acesso aos dados pode ser semelhante ao acesso cíclico à memória, evidenciando um dos problemas da utilização do LRU neste tipo de aplicação. Consequentemente, a utilização deste algoritmo pelo Spark para o gerenciamento da memória pode acarretar em degradação no desempenho do *framework*.

1.2 OBJETIVOS DO TRABALHO

Buscando alcançar um desempenho superior no processamento de aplicações com reutilização de dados, este trabalho tem como objetivo o desenvolvimento de um modelo de Gerenciamento Dinâmico da Memória no Spark. Por meio deste modelo, busca-se extrair métricas da execução da aplicação e dos RDDs utilizados por esta, de modo a utilizar estas informações para realizar remoção dos dados em *cache*, bem como decidir o momento em que essas operações devem ser executadas.

Para tanto, o modelo de Gerenciamento proposto é dividido em dois componentes: um algoritmo de gerenciamento das partições de RDDs armazenadas em memória e um agente de monitoramento responsável por obter informações sobre a execução de aplicações.

O algoritmo utilizado para realizar o gerenciamento da memória implementada por este modelo visa decidir quais blocos devem ser removidos da memória em situações onde a mesma encontra-se sobrecarregada. Tal implementação é baseada no algoritmo LRU e consiste em combinar a frequência de reutilização, extraída a partir dos RDDs identificados na aplicação, com a localidade temporal proporcionada pelo LRU.

O agente de monitoramento tem como objetivo prever a necessidade de remoção de partições da memória, por meio da instrumentação da execução de aplicações Spark. Este agente consiste em uma aplicação Java, responsável por acompanhar o andamento da execução da aplicação e a quantidade de memória utilizada para realizar o processamento da aplicação em questão.

A previsão da necessidade de liberar espaço em memória é realizada utilizando dois critérios: um *threshold* de ocupação da memória e o estado atual da execução da aplicação. O limiar de ocupação busca definir um limite máximo para a quantidade de memória ocupada, sendo inferior a 100%, de modo a garantir que haja espaço disponível para novos dados. O estado atual da execução da aplicação refere-se a quantidade de memória utilizada pela aplicação e a existência de computações pendentes para execução.

Assim, cenários onde a memória ocupada é superior ao *threshold* e ainda há computações pendentes indicam a possível necessidade de liberação de espaço na memória. Uma vez identificada esta necessidade, o agente de monitoramento envia uma notificação ao nodo Spark cuja memória encontra-se sobrecarregada, para que este nodo execute a remoção de partições da memória. Esta notificação é feita através do Apache ZooKeeper.

A validação do modelo para Gerenciamento Dinâmico da Memória é realizada por meio de experimentos, visando avaliar o tempo de execução dos *benchmarks*, visto que o tempo utilizado para processar as aplicações apresenta o maior impacto quando alterada a política de gerenciamento de memória. Através desta avaliação, compara-se o modelo proposta ao algoritmo LRU implementado nativamente pelo Spark em versões 2.x, uma vez que estas versões apresentam diferenças no gerenciamento da memória utilizada.

Os experimentos foram realizados com *benchmarks* focados na reutilização de dados em computações futuras. Para isto, utilizou-se as aplicações *PageRank*, *Logistic Regression* e *K-*

Means, visto que cada aplicação apresenta um comportamento diferente durante sua execução.

1.3 ESTRUTURA DO TRABALHO

O restante do trabalho encontra-se organizado da seguinte maneira: no Capítulo 2 são descritos os conceitos fundamentais acerca das ferramentas utilizadas no desenvolvimento do trabalho. Além disso, neste Capítulo é apresentado e discutido os trabalhos relacionados a este, encontrados na literatura.

O Capítulo 3 é destinado a descrever o modelo de Gerenciamento Dinâmico proposto por este trabalho. Para tanto, é demonstrando o desenvolvimento do trabalho juntamente com os detalhes de funcionamento e implementação, bem como o processo de experimentação e os resultados obtidos. Por fim, o Capítulo 4 apresenta as considerações deste trabalho.

2 REFERENCIAL TEÓRICO

Este Capítulo apresenta a fundamentação teórica relativa a este trabalho. Para tanto, o Capítulo encontra-se organizado da seguinte maneira: na Seção 2.1 é explicado o *framework* Apache Spark juntamente com o modelo de memória utilizado. A Seção 2.2 é dedicada à apresentação do Apache ZooKeeper. Por fim, a Seção 2.3 tem como objetivo explanar trabalhos relacionados a este, encontrados na literatura.

2.1 APACHE SPARK

O Spark é um *framework* de código aberto, projetado com o objetivo de realizar o processamento de dados massivos de maneira paralela e distribuída. A criação de aplicações Spark é realizada através de uma API (*Application Program Interface*) disponibilidada pelo *framework*.

Através desta API, é possível criar e manipular RDDs (KARAU et al., 2015), utilizando transformações e ações. As transformações são métodos que recebem um RDD como entrada e geram um ou mais RDDs como saída. São exemplos de transformações os métodos *map* e *filter*. Enquanto o método *map* aplica uma mesma função para todos os dados do *dataset* contido no RDD, o método *filter* realiza um filtro neste conjunto de dados, obedecendo uma condição definida pelo desenvolvedor.

As ações são operações cujo objetivo é computar o RDD e gerar um valor, como é caso dos métodos *reduce* e *count*. O método *reduce* agrega o conjunto de dados do RDDs em um único valor. Já o método *count* realiza uma contagem do número de registros armazenados no RDD.

Juntamente com os métodos de transformações e ações, são disponibilizados ao desenvolvedor métodos auxiliares a fim de realizar o armazenamento de RDDs, permitindo que estes sejam mantidos em *cache* após sua computação. A possibilidade de realizar a persistência dos dados evita que o RDD seja recomputado a cada reutilização em computações futuras. Neste sentido, dois métodos podem ser utilizados: *cache()* e *persist()*. O método *cache()* armazena os dados em memória principal, sendo este o nível padrão para armazenamento dos dados em *cache*. Por outro lado, o método *persist()* permite que seja especificado o nível de armazenamento, isto é, memória, disco ou combinação de ambos, para manter as partições em *cache*.

Internamente, um RDD é composto por:

a) uma lista de partições responsáveis por armazenar os dados. Geralmente, uma partição do RDD corresponde a um bloco de dado armazenado no HDFS (*Hadoop Distributed File System)*, com tamanho padrão de até 128 MB para o Hadoop (versão 2);

b) uma função para computar cada partição;

b) uma lista de dependência de outros RDDs;

b) opcionalmente, um particionador para RDDs chave-valor (isto é, hash-particionado) e;

b) uma lista de locais preferidos no *Cluster* para realizar a computação de cada partição, sendo este atributo opcional.

A avaliação dos RDDs criados é realizada de forma *lazy*, ou seja, sua computação não é efetuada no momento de sua criação, sendo processados apenas quando for necessário. Quando um novo RDD é gerado, este carrega consigo uma referência do(s) RDD(s) ancestral(is), adicionado(s) a sua lista de dependências. Tal processo é realizado na criação de qualquer RDD, gerando o grafo direcionado acíclico de dependências da aplicação.

O grafo direcionado acíclico (DAG – *Directed Acyclic Graph*), denominado de *lineage* do RDD, é utilizado para realizar a computação do RDD juntamente com suas respectivas dependências. Este grafo constitui um importante mecanismo de recuperação de falhas implementado pelo Spark. Assim, em situações onde partições são perdidas, o sistema pode realizar a recomputação dos dados a partir de uma réplica restante dos dados de entrada (KARAU et al., 2015).

Ao ser aplicada uma ação a um determinado RDD, este é submetido ao escalonador do Spark, denominado *DAGScheduler*, com o objetivo de transformar a sua *lineage* em um plano de execução físico composto por estágios. Para a criação destes estágios, o *DAGScheduler* particiona o grafo do RDD em limites *shuffle*, isto é, em limites onde há redistribuição dos dados entre partições.

Dependências onde não há redistribuição de dados são enfileiradas em um mesmo estágio para serem computadas, como é o caso das transformações *map* e *filter*. Em contrapartida, dependências *shuffle* requerem dois estágios para serem computadas, sendo o primeiro estágio para realizar a escrita dos arquivos de saída e um segundo estágio para ler estes dados, após uma barreira de sincronização entre as operações, (SPARK, 2017a).

O procedimento de criação do plano de execução da aplicação, denominado de *job*, é realizado sempre que uma ação é aplicada a um determinado RDD. Assim, a execução de uma mesma aplicação pode ser dividida em diversos *jobs* até a sua completa finalização.

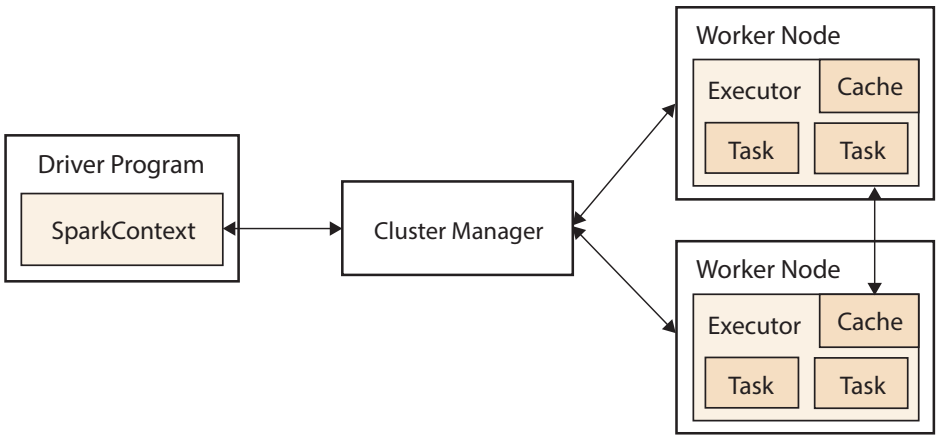
Após ser gerado o plano de execução físico pelo *DAGScheduler*, o Spark executa os estágios da aplicação distribuindo o processamento. Esta execução é feita através de *tasks*. Uma *task* consiste na menor unidade de computação possível, onde cada uma destas representa uma computação responsável por executar a mesma fatia de código, mas em diferentes porções dos dados da aplicação. (KARAU; WARREN, 2017).

2.1.1 Arquitetura do *Framework*

Para realizar o processamento de aplicações, o Spark utiliza o modelo *Master* e *Worker* (Mestre e Trabalhador) para seu funcionamento do *cluster*. Deste modo, sua arquitetura é composta basicamente por três componentes: *Driver Program*, *Cluster Manager* e *Worker*, conforme ilustra a Figura 1.

O *Driver Program* é responsável por hospedar o contexto da aplicação em um processo na *Java Virtual Machine* (JVM), sendo contido no nodo *Master* do *Cluster*. Além disso, o *Driver* é responsável por armazenar o escalonador do Spark (DAGScheduler) e gerenciar os *jobs* da aplicação Spark.

Figura 1 – Arquitetura do Apache Spark.



Fonte: Adaptado de (FRAMPTON, 2015, p. 8)

O *Cluster Manager* é encarregado de gerenciar as máquinas que serão utilizadas como *Workers* durante o processamento das aplicações Spark. De acordo (CHAMBERS; ZAHARIA,

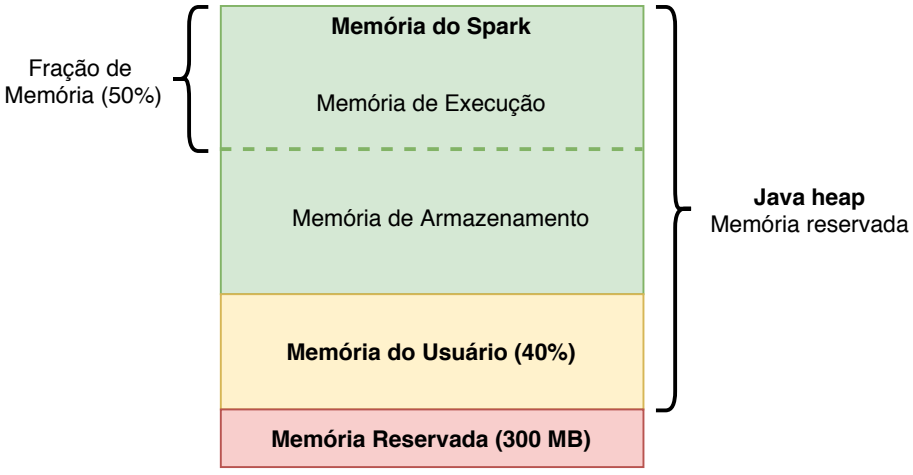
2018), são esses gerenciadores de *Clusters* que concedem recursos para completar a execução da aplicação. Por padrão, o Spark inclui um gerenciador de recursos *stand-alone* incluso na distribuição do *framework*. Além deste, há suporte para o Apache Hadoop YARN[2], Apache Mesos[3] e Kubernetes[4].

Os *Workers* são os nodos que realizam o processamento das *tasks* necessárias para a computação do *job*, sendo estes nodos os encarregados de armazenar os *Executors* do Spark. Um *Executor* consiste em um processo JVM isolado lançado pelo Spark, com o objetivo executar o código delegado pelo *Driver Program* e reportar seu *status*. Além disso, cada *Executor* possui sua região de memória isolada capaz de prover o armazenamento em memória para RDDs. O gerenciamento deste espaço é feito de forma autônoma pelo próprio *Executor*, utilizando o algoritmo LRU.

2.1.2 Modelo de Divisão da Memória

O espaço de memória alocado e gerenciado por cada *Executor* pode ser dividio em três principais regiões de memória: Reservada, Usuário e Spark. Esta divisão pode ser observada na Figura 2.

Figura 2 – Arquitetura da Memória do Apache Spark.



Fonte: Adaptado de (SPARK, 2017b)

A Memória Reservada é uma região alocada estaticamente, via código fonte, com tamanho igual a 300MB, não sendo incluída no cálculo de memória disponível no Spark. O principal

[2] Disponível em: https://hadoop.apache.org
[3] Disponível em: http://mesos.apache.org/
[4] Disponível em: https://kubernetes.io/

objetivo dessa área de memória é armazenar objetos internos do Spark.

A Memória do Usuário é a fração do *Java heap* disponível para o usuário utilizar. Nessa região são armazenadas as estruturas, objetos e dados criados durante o desenvolvimento de aplicações Spark. O tamanho desta região corresponde a 40% da memória disponível após a alocação da memória reservada. É importante ressaltar que o gerenciamento desta memória não é realizado pelo Spark, sendo delegado inteiramente ao *Garbage Collector* (GC) da JVM.

A Memória do Spark é a região efetivamente controlada e gerenciada pelo Spark e GC da JVM. O tamanho desta região corresponde à fração de 60% da memória disponível do *Java heap* após a alocação da Memória Reservada. Esta região é subdividida em duas porções: Memória de Execução e Memória de Armazenamento, cada qual com 50% do tamanho da área inicial da Memória do Spark.

A Memória de Execução é responsável por armazenar os objetos e dados resultantes de operações intermediárias durante a execução de *tasks*. Já a Memória de Armazenamento abriga os dados computados e assinalados para serem mantidos em memória, espaço para a deserialização de dados e armazenamento das variáveis de *broadcast*. Estas variáveis têm como objetivo compartilhar o estado da execução do *job* entre os nodos do *cluster*.

A forma em que a divisão do *Java heap* é feita variada de acordo com a versão do Spark. Assim, em versões 1.5.x e anteriores do Spark, a divisão de memória era realizada de forma estática, isto é, alocava-se 50% do espaço de memória para cada subdivisão, independentemente da necessidade. A partir da versão 1.6 foi implementado o balanceamento dinâmico da divisão de memória, permitindo que a quantidade de memória disponível em cada subespaço seja alocada de acordo com as necessidades no momento da execução.

Se durante a execução de uma determinada tarefa não houver memória suficiente disponível na Memória de Execução, o Spark pode alocar a memória extra necessária no espaço livre da Memória de Armazenamento. Caso a Memória de Armazenamento esteja esgotada, blocos desta região podem ser removidos a fim de disponibilizar espaço para a conclusão da tarefa em questão.

Em casos onde há necessidade de armazenar novos dados na Memória Armazenamento e o espaço desta região estiver esgotado, o Spark pode remanejar uma porção livre da Memória de Execução para o armazenamento. Caso não seja possível realizar esta operação, o Spark poderá remover blocos da Memória de Armazenamento a fim de disponibilizar espaço. A remoção desses blocos de memória é realizada seguindo a ordem definida pelo algoritmo LRU.

Embora a divisão da memória alocada para o Spark seja determinada em tempo de execução, o espaço de memória total ocupado pela *Java heap* pode ser ajustado. Cada aplicação Spark em execução pode ter uma configuração de memória associada, independente das demais. Estas configurações são metadados da aplicação bem como características de execução (CHAMBERS; ZAHARIA, 2018), contendo informações que serão utilizadas pelo Spark para configuração do ambiente de execução.

Por padrão, o Spark busca um arquivo denominado *spark-defaults.conf*, responsável por armazenar as configurações no formato chave-valor. Através deste arquivo, configurações como tamanho da memória utilizada pela *Java heap* e percentual de divisão do espaço entre a Memória de Armazenamento e a Memória de Execução, dentre outras, podem ser ajustadas pelo usuário.

Durante a execução de aplicações, é possível monitorar a quantidade de memória utilizada, assim como o *status* da execução da aplicação, através de uma API REST (*Representational State Transfer*). Por meio deste serviço, os usuários podem monitorar aplicações em execução, além de permitir a realização de construções de visualizadores customizados (GULATI, 2017).

2.1.3 Gerenciamento da Memória Disponível

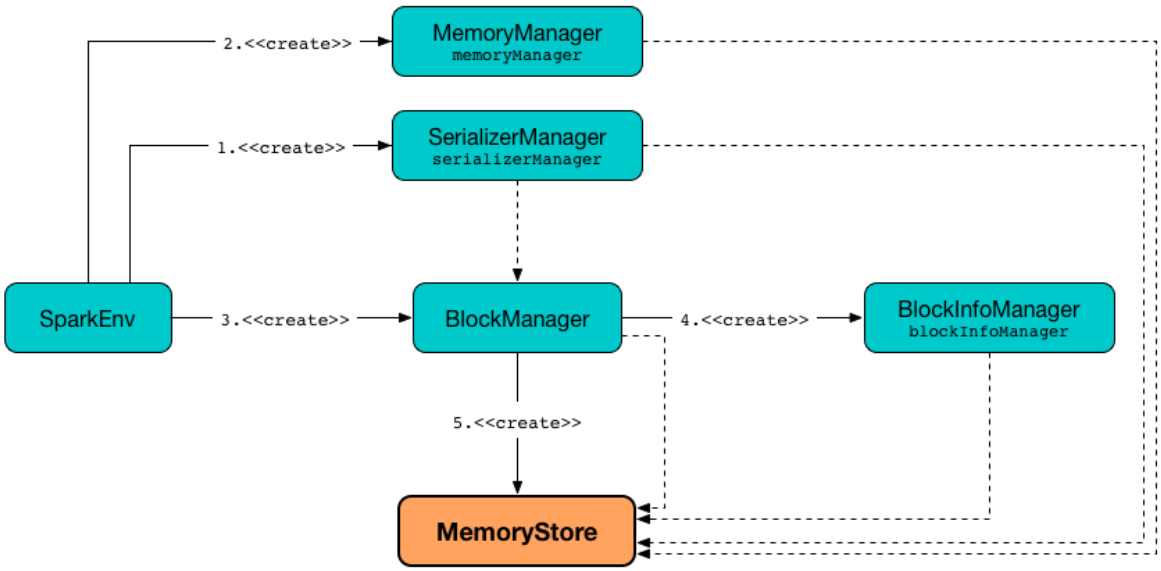
O Spark é um *framework* para processamento de grandes volumes de dados em memória principal. Consequentemente, exige um bom gerenciamento da memória disponível, a fim de otimizar sua ocupação e alcançar melhor desempenho. De forma geral, o gerenciamento da memória realizado pelo Spark é composto por três classes principais: *MemoryManager*, *BlockManager* e *MemoryStore*. O diagrama de inicialização destas classes é exibido na Figura 3.

A classe *SparkEnv* é responsável por armazenar as configurações globais do *cluster* e compartilhar essas informações com todos os nodos Spark. A partir desta classe, cria-se as classes *MemoryManager*, *SerializerManager* e *BlockManager*.

A classe *MemoryManager* visa oferecer uma abstração da memória, de modo a compartilhar o espaço disponível entre as porções de Memória de Armazenamento e Memória de Execução. Assim, existe uma instância dessa classe em execução em cada *Executor* ativo no *cluster* Spark.

A classe *SerializerManager* é responsável por configurar o serializador utilizado pelo

Figura 3 – Fluxo de Inicialização do Gerenciamento de Memória do Spark



Fonte: (LASKOWSKI, 2017)

Spark, incluindo aquele utilizado nas operações de *shuffle*. Esta classe é utilizada para a criação do *BlockManager* e do *MemoryStore*.

A classe *BlockManager* oferece interfaces para realizar a adição e recuperação de blocos de dados localmente no *Executor* e remotamente, isto é, em outros nodos do *cluster* (SPARK, 2017c). A adição e recuperação desses blocos pode ser realizada em diferentes fontes de armazenamento, tais como memória principal e armazenamento estável.

O *BlockManager* é responsável também, pela criação do *BlockInfoManager* a qual tem como objeto rastrear os metadados dos blocos de dados de maneira individual, os quais são utilizados para informar a localização do bloco, isto é, memória ou disco. Ainda, é função do *BlockInfoManager* fornercer os *locks* de leitura e escrita desses blocos.

Na classe *MemoryStore* encontram-se os métodos e variáveis para armazenamento e remoção de informações da memória (SPARK, 2017d). Estas informações podem ser *Arrays* de objetos Java desserializados ou *Buffers* de *bytes* serializados. Além disso, cabe a esta classe implementar a política de gerenciamento de memória utilizada pelo Spark: o algoritmo LRU.

O LRU é um algoritmo clássico para gerenciamento de memória em diversos sistemas. No Spark, a implementação deste algoritmo é realizada utilizando uma lista encadeada (*LinkedHashMap*) provida nativamente pela distribuição Java. Através desta estrutura, o Spark mantém uma estrutura de lista encadeada contendo os dados de maneira ordenada por acesso.

Cada vez que uma determinada posição dessa lista é acessada, esta posição é movida para o final da lista. Por consequência, as posições mais recentemente acessadas são encontradas ao final desta lista encadeada.

A estrutura de lista é iniciada com 32 posições e um fator de carregamento de 75%. Isso significa que, ao atingir 75% da capacidade de ocupação, novas posições começam a ser alocadas dinamicamente pela JVM. Uma particularidade dessa lista está relacionada à forma como a mesma deve ser manipulada em ambientes *multithread*. Nesses ambientes, o acesso deve ser feito de maneira sincronizada entre as *threads* em execução, pois a implementação não é realizada de maneira *thread-safe*. Essa sincronização busca manter a ordenação, uma vez que há alteração nas posições armazenadas na lista quando ocorre acesso a mesma.

A sincronização entre as *threads* em execução no Spark é realizada através de um *lock* global do objeto. Deste modo, apenas uma *thread* consegue realizar a escrita, leitura ou remoção de um bloco de dado a partir da estrutura, evitando condições de corrida (*race condition*) entre as *threads*. Entretanto, esta sincronização pode acarretar em uma sobrecarga para a aplicação, uma vez que durante essas operações apenas uma *thread* consegue manipular a estrutura, bloqueando as demais.

Cada Spark *Worker* executando no *cluster* possui suas próprias estruturas para controlar as regiões de memória. Assim, cada nodo Spark gerencia sua memória de forma autônoma e sem dependências externas. Além disso, devido à distribuição dos dados do RDDs entre os nodos, o momento em que blocos de memória deverão ser removidos pode variar entre diferentes nodos.

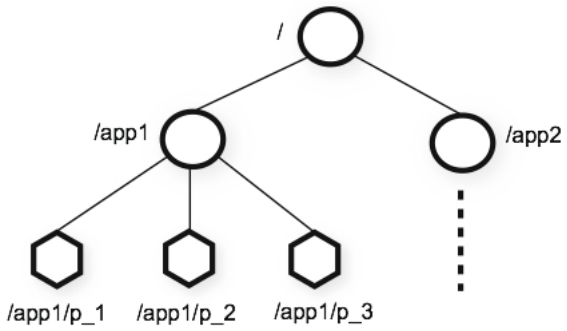
2.2 APACHE ZOOKEEPER

O Apache ZooKeeper é um *framework* de código aberto cujo objetivo é prover soluções para problemas de coordenação em grandes conjuntos de nodos (HUNT et al., 2010). Através deste, é possível coordenar grupos de nodos entre si e manter o compartilhamento de informações com técnicas de sincronização robustas.

O ZooKeeper realiza o processo de coordenação através de um *namespace* hierárquico compartilhado organizado de maneira semelhante a um sistema de arquivos padrão. Entretanto, diferente dos sistemas de arquivos, os nodos responsáveis por armazenar os dados, denominados de *znodes*, são estruturas similares a arquivos e diretórios projetados para serem armazenados em memória principal. Um *namespace* do *znode* pode conter dados associados e referências a

outros *znodes*. A Figura 4 exibe um nodo denominado de *"/"*, o qual possui dois outros nodos associados, exemplificando a arquitetura utilizada no *namespace*.

Figura 4 – Estrutura de *namespace* do Apache ZooKeeper



Fonte: (HALOI, 2015)

Um *znode* mantém uma estrutura do estado atual dos dados armazenados, a qual inclui o número da versão, o *Action Control List* (ACL) e um *Timestamp*. O número da versão representa a quantidade de vezes que os dados armazenados pelo *znode* foram alterados. O ACL é o mecanismo de autenticação, assegurando quais nodos do *cluster* podem realizar operações de escrita e leitura. O *Timestamp* descreve o tempo decorrido desde a criação do nodo até o momento da modificação dos dados armazenados.

Uma importante característica do ZooKeeper é a capacidade de oferecer suporte a *Watchers*, permitindo que clientes possam receber notificações ocorridas em um determinado *znode*. A manutenção da comunicação entre o *Master* e os clientes é realizada através de mensagens *heartbeat*, as quais garantem que a conexão com os clientes entra-se ativa.

Uma mudança em um *znode* consiste em uma modificação nos dados associados a este nodo ou uma alteração em um de seus nodos filhos (JUNQUEIRA; REED, 2013). Através dos *znodes*, as aplicações podem oferecer suporte a serviços de sincronização, fila de mensagens, além de gerenciamento de configuração do *cluster*.

Juntamente com seu sistema de *znodes*, o ZooKeeper oferece uma API com suporte oficial para as linguagens Java e C, a fim de realizar a criação de aplicações cliente. Através destes clientes, pode-se conectar, manipular dados associados aos *znodes*, coordenar operações e interagir com ZooKeeper.

2.3 TRABALHOS RELACIONADOS

O algoritmo utilizada para o gerenciamento do espaço disponível pode afetar diretamente o desempenho do *framework* durante o processamento de aplicações. Nesse sentido, alternativas têm sido pesquisadas e desenvolvidas buscando aumentar a eficiência do Apache Spark.

O trabalho desenvolvido por (WANG; ZHANG; GAO, 2015) apresenta uma abordagem para analisar o comportamento do Spark ao realizar acessos à memória visando melhorar a ordem da execução de ações nos RDDs. Para tanto, os autores utilizam um algoritmo guloso para encontrar a melhor sequência para executar as ações aplicadas aos RDD, uma vez que o processamento de uma ação independe das demais. Juntamente com este algoritmo, a abordagem tem como objetivo calcular um peso do RDD em uma *task*. O cálculo deste peso considera 6 critérios: tamanho dos dados de entrada do RDD, custo de computação, frequência de reutilização, tamanho do conjunto de dados de saída da transformação aplicada, ciclo de vida do RDD (isto é, por quanto tempo este é manipulado) e custo de armazenar o RDD em disco. Em seguida, é apresentado um novo algoritmo denominado de ASRD (*Action's Sequence and RDD Weight*), o qual combina a análise da execução aliada ao peso do RDD calculado. Os autores não especificam a versão do Spark utilizada, tampouco o ambiente usado para validação do método proposto. Entretanto, os autores do trabalho não apresentaram resultados em termos percentuais, limitando-se a afirmar que a metodologia proposta obteve um ganho na taxa de acerto de acesso à memória, quando comparado ao LRU.

Em (DUAN et al., 2016) é apresentado um algoritmo para a seleção de RDDs a serem armazenados em *cache* baseado na frequência de utilização dos RDDs na aplicação. Dessa forma, remove-se a necessidade do desenvolvedor de decidir qual RDD deve ser mantido em *cache* para ser reutilizado posteriormente. Quando a memória alcança o seu limite disponível, blocos são removidos de acordo com uma nova política proposta pelos autores: o algoritmo *Weight Replacement*. Esse algoritmo consiste na atribuição de um peso, calculado a partir do custo de recomputação da partição, além da frequência de utilização e tamanho da partição. A proposta apresentada foi implementada no Spark 1.1.0 e validada utilizando 17 *datasets* reais. De acordo com os autores, o algoritmo proposto obteve melhores resultados quando comparado ao LRU, reduzindo o tempo de execução da aplicação *PageRank*. Entretanto, não são apresentados percentuais deste ganho de desempenho.

Em (ZHANG et al., 2017), os autores apresentam uma estratégia inteligente para *checkpointing* e remoção de blocos da memória pelo Spark. Tal estratégia consiste na atribuição de uma prioridade *P* para cada RDD, calculada a partir do tempo de recomputação, do tempo de *checkpointing* e do grau de dependência entre os RDDs. Assim, quanto maior o valor de *P*, maior deve ser a prioridade do RDD permanecer em memória. A validação desta proposta foi realizada utilizando o Spark 1.5 em 4 servidores, com a memória variando entre 5G, 10G e 20G. De acordo com os autores do trabalho, o resultado obtido demonstra um ganho médio de desempenho de 13,63% quando comparado ao LRU.

O trabalho realizado por (XU; SAXENA; CHIU, 2016) apresenta um gerenciador de *cache* distribuído denominado de *Neutrino*. Esse gerenciador é responsável por tomar decisões relativas ao armazenamento dos RDDs com base em configurações do *cluster* e na execução da aplicação. Para tanto, o *Neutrino* analisa a ordem em que os RDDs serão utilizados e calcula o menor tempo de execução possível do *Job*. Esse gerenciador foi implementado como uma extensão do Spark 1.5.0. De acordo com os autores, a solução proposta apresenta resultados promissores quanto a sua utilização.

Em (GENG et al., 2017) é demonstrada uma estratégia denominada *Least Cost Strategy* (LCS) para a remoção de blocos de memória, a qual integra o custo de recuperação da partição em cache e a lógica de execução da aplicação. Este algoritmo combina o tempo necessário para criar, remover e recomputar cada partição dos RDDs juntamente com uma frequência de reutilização de cada partição. A implementação dessa estratégia foi realizada utilizando o Spark 1.5.2. Conforme os autores, o LCS consegue um desempenho em torno de 30% melhor que o LRU.

Em (YANG et al., 2018), os autores apresentam um algoritmo para determinar e armazenar adaptativamente os *datasets* intermediários mais valiosos, isto é, aqueles que poderão ser reutilizados no futuro. Assim, este algoritmo automatiza a decisão de quais RDDs devem ser persistidos em memória, analisando o DAG gerado pelo Spark durante a execução do *job*. O algoritmo foi implementado utilizando o Spark 2.2.1 e sua validação ocorreu utilizando a aplicação *Ridge Regression*. Segundo os autores, o algoritmo proposto consegue reduzir em 12% o trabalho necessário para realizar as recomputações dos RDDs removidos.

No trabalho desenvolvido por (INAGAKI et al., 2018), é apresentada uma metodologia de *cache* adaptativa para o Spark, a qual modifica o comportamento dos métodos de armazenamento e remoção de dados da *cache*, dependendo das características da aplicação. Em situações

onde os dados da aplicação são frequentemente movidos da memória para o disco, esta abordagem de *cache* adaptativo move estes dados diretamente para o armazenamento estável. Além disso, se estes dados armazenados no disco são requeridos pela aplicação, o mecanismo de *cache* adaptativo carrega os dados necessários diretamente para a Memória de Execução do Spark. Estas ações de carga de *cache* são realizadas analisando a execução da aplicação ao invés das dependências entre os RDDs. A implementação desta estratégia foi realizada utilizando a versão 2.0.1 do Spark. De acordo com os autores, este mecanismo de gerenciamento de *cache* reduziu em 33% o tempo de execução do *benchmark* K-Means, quando comparado ao LRU utilizando o nível de armazenamento Memória e Disco nativo do Spark.

É possível encontrar na literatura algoritmos e soluções que têm como objetivo melhorar a forma na qual o Spark realiza o processamento de aplicações onde há a reutilização de dados em computações futuras, uma vez que o algoritmo LRU pode acarretar em degradação no desempenho do *framework* nesta classe de aplicações. Entretanto, quando analisamos tais soluções, duas características ficam evidentes: versão do Spark e metodologia utilizada.

A primeira característica está relacionada à versão utilizada para implementação e validação, onde geralmente os trabalhos concentram-se em versões defasadas e com um modelo de memória diferente daquela utilizada por versões mais novas do Spark. A segunda característica encontrada em trabalhos disponíveis na literatura relaciona-se com as soluções propostas, onde comumente foca-se em algoritmos estáticos e reativos. Embora as soluções encontradas utilizem métricas colhidas a partir da aplicação, estas soluções tendem a executar apenas quando há necessidade de liberação de memória.

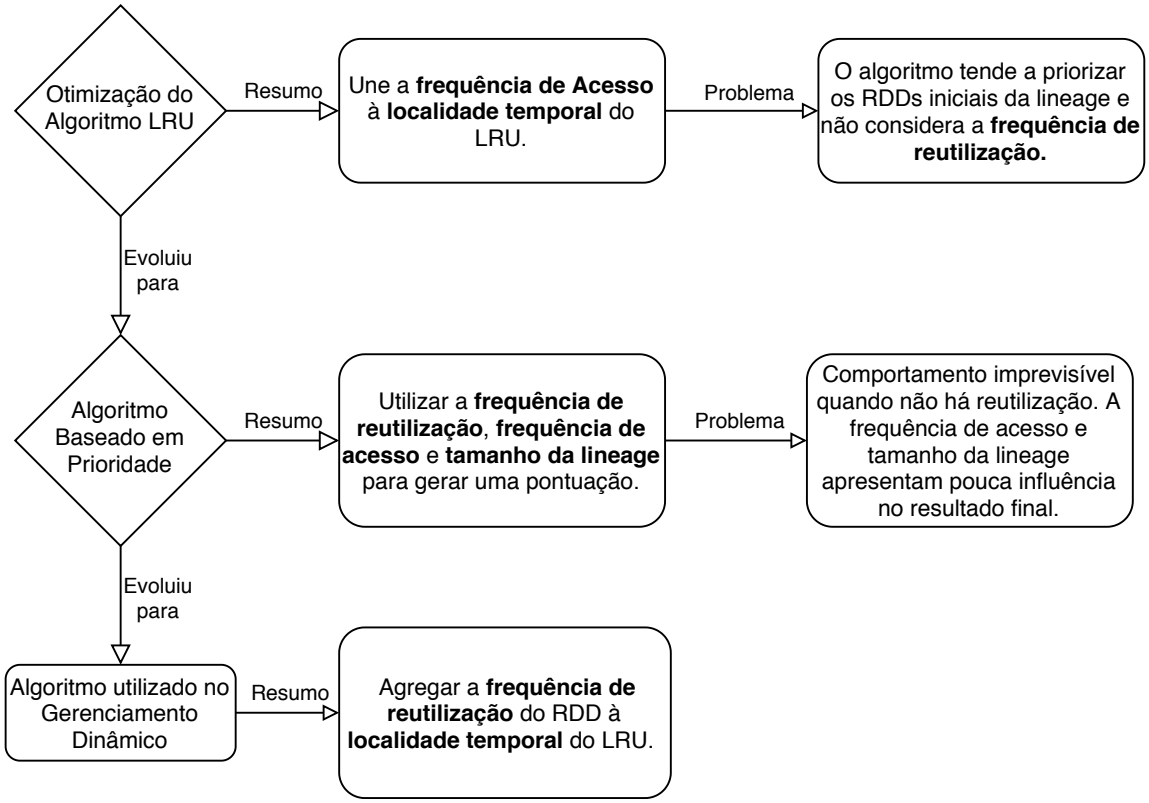
Diferentemente dos algoritmos e soluções encontradas na literatura, este trabalho tem como objetivo implementar um modelo para realizar o Gerenciamento Dinâmico da Memória utilizada pelo Spark, em versões 2.x do *framework*. Esta solução tem como foco as aplicações onde há reutilização de dados em computações futuras. Para tanto, o modelo proposto é composto por dois componentes: um algoritmo de gerenciamento de memória e um agente de monitoramento.

O algoritmo de gerenciamento de memória visa extrair a frequência de reutilização dos RDDs da aplicação e combiná-la com o algoritmo LRU. Já o agente de monitoramento tem como objetivo obter o *status* e informações da aplicação em execução e decidir se deve ocorrer a remoção dos dados em *cache*. Através deste modelo, busca-se corrigir as deficiências encontradas no algoritmo LRU no processamento de aplicações onde há reutilização de dados.

3 GERENCIAMENTO DINÂMICO DA MEMÓRIA

A concepção do Gerenciamento Dinâmico da Memória em Aplicações com Reuso de Dados no Apache Spark proposto por este trabalho foi realizada de forma modular. Deste modo, duas implementações foram previamente desenvolvidas até culminar na versão final do algoritmo utilizado para gerir as partições de RDDs em memória na implementação do Gerenciamento Dinâmico. A Figura 5 ilustra, de forma resumida, a evolução do trabalho, descrevendo o funcionamento de cada algoritmo e as razões para sua alteração.

Figura 5 – Fluxo Evolutivo do Algoritmo de Gerenciamento Dinâmico da Memória.



Fonte: Próprio autor.

O primeiro algoritmo desenvolvido, a Otimização do Algoritmo LRU, tinha como objetivo agregar a Frequência de Acesso do RDD juntamente com a ordenação temporal provida pelo LRU. A Seção 3.1 dedica-se a descrever o funcionamento deste algoritmo, junto com sua respectiva implementação. Ainda nesta Seção, são detalhados os experimentos realizados e os resultados obtidos.

A partir dos problemas identificados na Otimização do Algoritmo LRU, um segundo

algoritmo foi desenvolvido, sendo denominado de Algoritmo Baseado em Prioridades. Este algoritmo visava unir a Frequência de Acesso, o Tamanho da *Lineage* e a Frequência de Reutilização dos RDDs a fim de gerar uma pontuação, simbolizando a prioridade deste RDD estar em memória. Detalhes relativos ao seu funcionamento, implementação, experimentos e resultados são explanados na Seção 3.2

Utilizando como base os dois algoritmos previamente desenvolvidos, uma nova otimização do algoritmo LRU foi desenvolvida com o objetivo de corrigir as deficiências do LRU e dos algoritmos anteriores. Esta otimização tem como finalidade agregar a localidade temporal do LRU à Frequência de Reutilização dos RDDs manipulados no *job*, sendo este algoritmo utilizado no **Gerenciamento Dinâmico da Memória**. A Seção 3.3 é destinada a expor este algoritmo, juntamente com os dois componentes utilizados para implantar o Gerenciamento Dinâmico da Memória, descrevendo as nuances relativas ao funcionamento e implementação no Spark. A Seção é finalizada descrevendo os experimentos e resultados obtidos pelo Gerenciamento Dinâmico.

3.1 OTIMIZAÇÃO DO ALGORITMO LRU

O algoritmo LRU utilizado nativamente pelo Spark pode ocasionar uma possível degradação no desempenho da ferramenta no processamento de aplicações com reutilização de dados. Visando sanar esta deficiência, uma primeira modificação na política de gerenciamento de memória foi implementada, a qual consistia em uma otimização do algoritmo LRU utilizado pelo *framework*.

A otimização desenvolvida consiste em um algoritmo que combina a **localidade temporal** do LRU com a **frequência de acesso** dos RDDs utilizados na computação do *job*. A frequência de acesso tem como objetivo descrever a quantidade de vezes a qual cada RDD é utilizado no processamento da aplicação.

Para tanto, este algoritmo analisa o *job* gerado pelo *DAGScheduler* a fim de identificar os RDDs manipulados e obter suas respectivas frequências de acesso. A partir destas informações, mantém-se uma lista de partições dos RDDs ordenadas pela frequência de acesso. Uma vez realizada esta ordenação, conforme as novas partições dos RDDs são carregadas para a memória principal, estas partições são armazenadas de maneira ordenada pela frequência de acesso, de maneira a manter a ordenação inicial.

Em cenários 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 e, em caso de empate, é removida a partição onde o acesso ocorreu há mais tempo.

A implementação deste algoritmo implica na obtenção da frequência de acesso de cada RDD da aplicação e na difusão desta informação entre todos os *Executors* da aplicação. As frequências de acesso dos RDDs são obtidas através do escalonador do Spark, o *DAGScheduler*, antes de iniciar a execução da aplicação. Para isso, navega-se recursivamente entre o plano de execução para a computação do *job* gerado pelo escalonador, com o objetivo de identificar os RDDs utilizados. O resultado desta etapa consiste em uma lista contendo todos os RDDs manipulados no *job*. Em seguida, a lista gerada é utilizada para obter a frequência de acesso de cada RDD, analisando a *lineage* de cada RDD.

Ao final do processo de análise da *lineage*, uma estrutura de *HashMap* é gerada, mapeando cada RDD do *job* com sua respectiva frequência de acesso. Assm, o consumo extra de memória será a quantidade máxima de entradas nesta estrutura de *HashMap*, igual ao número de RDDs manipulados na aplicação. Cada entrada desse *HashMap* possui dois inteiros, onde o primeiro inteiro corresponde ao identificador do RDD e o segundo a frequência de acesso deste mesmo RDD.

Após a obtenção das frequências de acesso, o próximo passo consiste em difundir esta lista contendo as frequências de acesso entre os *Executors* do *cluster*. A etapa de difusão se faz necessária uma vez que apenas através do *Driver* da aplicação, o qual hospeda o *DAGScheduler*, é possível obter informações referentes à frequência de acesso a cada RDD do *Job*.

A difusão é realizada de maneira síncrona entre todos os *Executors*. O sincronismo na comunicação entre o *Driver* e os *Executors* garante que os dados referentes aos RDDs estejam em todos os nodos antes de seu efetivo uso. Em contrapartida, a utilização deste tipo de comunicação acarreta em um atraso no início da execução. Isto ocorre devido ao comportamento das chamadas síncronas, as quais são bloqueantes, isto é, nenhum outro processamento é realizado até a conclusão da comunicação. Consequentemente, há uma sobrecarga na execução da aplicação, atrasando o início do processamento *job* em até 2 segundos, por comunicação realizada.

Esta otimização do algoritmo LRU teve como objetivo priorizar aqueles RDDs com maior frequência de acesso dentro do *job*, postergando sua remoção tanto quanto possível. Entretanto, conforme a *lineage* cresce, os RDDs iniciais tendem a apresentar uma alta frequência

de acesso, uma vez que estes RDDs são computados diversas vezes para originar novos RDDs.

Além disso, este algoritmo não considera um importante parâmetro para decidir quais partições remover: a frequência de reutilização do RDD. Este parâmetro tem relevância, uma vez que a remoção de um RDD com uma alta frequência de reutilização implica em sua completa recomputação a cada novo acesso, podendo implicar em uma degradação no desempenho do *framework*.

Ademais, quando analisadas as soluções encontradas na literatura, a frequência de reutilização de um RDD se mostra uma importante métrica para o planejamento de algoritmos focados em aplicação com reutilização de dados. Junto com a frequência de reutilização, métricas auxiliares podem ser agregadas, tais como custo computacional, a fim de tornar o gerenciamento da memória mais eficiente.

3.1.1 Experimentos e Resultados

A fim de validar o algoritmo desenvolvido, foram conduzidos experimentos em um ambiente distribuído visando avaliar seu impacto no tempo de execução das aplicações. Os experimentos foram realizados utilizando os algoritmos *PageRank* e *K-Means* como *benchmarks*. Estes *benchmarks* foram selecionados, uma vez que cada um deles apresenta um comportamento diferente no acesso à memória.

Uma característica comum aos dois *benchmarks* é a reutilização de dados em computações futuras. O *PageRank* visa classificar *links* de acordo com suas ligações, a fim de gerar uma relevância para um determinado *link*. O *K-Means* consiste em agrupar os dados de entrada em K grupos baseados em suas características, sendo este um algoritmo de *machine learning* não supervisionado. Em todos os experimentos, os *benchmarks* foram implementados pela suíte Intel HiBench (HUANG et al., 2010).

O ambiente utilizado foi a plataforma Grid'5000 (BOLZE et al., 2006), executando os *benchmarks* em um *cluster* com 5 nodos configurados da seguinte maneira: 1 Spark *Master*; 2 Spark *Workers*; 1 *HDFS Namenode* e 1 *HDFS Datanode*. Cada nodo do sistema era composto por um Intel Xeon X3440 @2.53GHz (4 *cores*/CPU) e 16GB de memória RAM, conectados via *ethernet* 1 Gbps. 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 *Executor* utilizou duas configurações de memória: 1GB, sendo 366,6 MB para a Memória e armazenamento e 2 GB, dos quais 912,3MB eram dedicados ao armazenamento de dados. Estas configurações de memória tinham

como objetivo estressar o gerenciamento do espaço disponível, a fim de verificar o impacto do algoritmo no tempo de execução das aplicações quando comparado ao LRU nativo do Spark. Por fim, os resultados obtidos são a média aritmética de 20 execuções.

Os resultados obtidos por este algoritmo são apresentados na Tabela 1. Estes resultados demonstraram um desempenho similar ao LRU, apresentando pouca variação no tempo de execução. Embora os tempos obtidos sejam semelhantes, a modificação do LRU é penalizada pela necessidade de comunicação entre os nodos do *cluster* para seu funcionamento. Analisando os *logs* gerados pela aplicação, cada comunicação síncrona pode levar até 2 segundos para ser executada, sendo este tempo diluído no tempo total de execução da aplicação.

Tabela 1 – Otimização do LRU: Tempos de Execução dos *Benchmarks*

Benchmark	Algoritmo	Memória Disponível	Tempo de Execução (em s)	Desvio Padrão (em s)
K-Means	LRU	1 GB	47,78	0,88
		2 GB	25,48	0,30
	Otimização do LRU	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
	Otimização do LRU	1 GB	176,93	3,43
		2 GB	152,78	1,85

A necessidade de efetuar comunicação síncrona entre o *Driver* da aplicação e os *Executors*, para cada *job* da aplicação, faz com que o processamento não seja iniciado até que a transmissão das informações para todos os nodos seja concluída. Esta comunicação é realizada independente da necessidade e, portanto, acarreta em um aumento do tempo de execução.

3.2 ALGORITMO BASEADO EM PRIORIDADES

Com o objetivo de resolver as deficiências identificadas na primeira otimização do algoritmo LRU (Seção 3.1), um novo algoritmo foi desenvolvido. Este algoritmo visa agregar três diferentes métricas, a saber: a Frequência de Reutilização, a Frequência de Acesso e o Tamanho da *Lineage* dos RDDs. O algoritmo busca determinar uma prioridade para cada RDD ser mantido em memória, sendo esta prioridade simbolizada por uma pontuação.

3.2.1 Descrição do Algoritmo

O algoritmo tem como objetivo obter métricas as quais serão utilizadas para gerir a memória do Spark. Assim, após a computação do plano de execução pelo *DAGScheduler*, o estágio resultante é utilizado para extrair métricas da aplicação e calcular a pontuação de cada RDD do *job*.

A partir do estágio gerado, navega-se recursivamente entre as dependências do *job*, a fim de identificar os RDDs manipulados. Na sequência, é realizada a extração das métricas Frequência de Reutilização, Frequência de Acesso e Tamanho da *Lineage*, a partir dos RDDs mapeados previamente.

Uma vez realizada a extração das métricas relativas aos RDDs do *job*, o próximo passo consiste em analisar os resultados. Nessa análise, são calculadas as pontuações dos RDDs utilizados na computação do *job*, simbolizando a prioridade deste RDD ser mantido em memória. A pontuação obtida é calculada a partir da frequência de reutilização multiplicada pela média aritmética da frequência de acesso somada ao tamanho da *lineage*, conforme exibe a Equação 3.1.

A Equação 3.1 foi concebida com a intenção de priorizar os RDDs com maior número de reutilizações dentro do *job*, em detrimento aos demais. Em situações onde não há reutilização do RDD, o algoritmo calcula uma pontuação para cada RDD baseando-se na média aritmética entre a frequência de acesso e o tamanho da *lineage*. O estudo da correlação da equação proposta é discutida na Seção 3.2.4.

$$P = n\_reusos * (0,5 * freq\_acesso + 0,5 * tam\_lineage) \tag{3.1}$$

Após concluídos o processamento e o cálculo das prioridades dos RDDs, o resultado final consiste em uma lista mapeando todos os RDDs com suas respetivas pontuações. Esta lista é armazenada no *Driver*. A difusão da lista com as pontuações de cada RDD é realizada entre os nodos do *cluster* sob demanda. Isso significa que a comunicação entre o *Executor* e o *Driver* irá ocorrer apenas quando a memória desse *Executor* estiver sobrecarregada e necessitando realizar a remoção de partições da memória.

Em caso de sobrecarga, o *Executor* irá solicitar a lista contendo a pontuação de cada RDD do *job*. Essa requisição ocorre em duas situações: quando a lista com as pontuações está vazia e quando o identificador de uma partição não é encontrado. O *status* de lista vazia significa

que o processamento da aplicação começou recentemente e nenhum bloco foi removido. A ausência de um identificador na lista indica que um novo *job* iniciou e novos RDDs foram manipulados, necessitando que a lista de pontuações seja atualizada no *Executor*.

3.2.2 Métricas utilizadas

O Algoritmo Baseado em Prioridade foi concebido com o objetivo de agregar três métricas e, a partir destas, determinar a ordem em que as partições de RDDs serão removidas. Assim, utiliza-se a Frequência de Acesso, o Tamanho da *Lineage* e a Frequência de Reutilização de cada RDD que será manipulado no processamento do *job*.

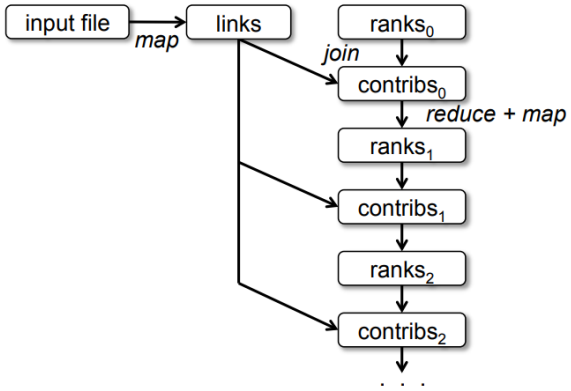
A **frequência de acesso** descreve o número de vezes que um RDD será utilizado na computação do *job*. Este número determina a quantidade de recomputações que serão realizadas utilizando o RDD caso este não seja armazenado em *cache*. Por exemplo, para a *lineage* $L = \{R_1 \rightarrow R_2 \rightarrow R_3\}$, se for aplicada uma ação aos RDDs R_2 e R_3, o R_1 será computado duas vezes, sendo a primeira para o subconjunto $SL = \{R_1 \rightarrow R_2\}$ e uma segunda vez para a *lineage L*.

O **tamanho da *lineage*** representa a quantidade de RDDs que devem ser previamente processados até que seja possível computar o RDD cuja ação foi aplicada. Dada a *lineage* $L = \{R_1 \rightarrow R_2 \rightarrow R_3\}$, o tamanho da *lineage* de R_3 é igual a 2, uma vez que deve-se necessariamente computar R_1 e R_2 até ser possível computar R_3. Salienta-se que, quanto maior o tamanho da *lineage* de um RDD, maior será o tempo exigido para recomputá-lo se necessário.

A **frequência de reutilização** refere-se ao número de vezes que o RDD será referenciado durante a computação do *job*. Assim, a obtenção do grafo de dependências visa identificar os RDDs com maior frequência de reutilização dentro do processamento. A Figura 6 exemplifica o grafo de dependências gerado durante a execução da aplicação *Pagerank*.

Analisando o grafo gerado na Figura 6, inicialmente é realizada a carga de um arquivo de entrada. A seguir, é aplicada uma função de *map* para gerar o RDD contendo os *links*. Durante a execução a aplicação, o RDD *links* é frequentemente reutilizado na geração de novos RDDs com resultados intermediários. Como consequência, este RDD torna-se um candidato a ser armazenado em *cache*, a fim de evitar sua recomputação a cada utilização e consequentemente reduzir o tempo de processamento da aplicação. Nesse exemplo, o RDD *links* possui uma frequência de reutilização igual a 3, enquanto os demais RDDs do *job* apresentam frequência igual a 1.

Figura 6 – Grafo de Dependências da Aplicação PageRank



Fonte: (ZAHARIA et al., 2012)

3.2.3 Implementação

Para realizar a implementação do Algoritmo Baseado em Prioridades, a primeira etapa do processo consiste na extração de métricas da aplicação a partir dos RDDs que serão utilizados no *job*. Para tanto, parte-se do RDD cuja ação foi aplicada, até não encontrar novas dependências. Na sequência, navega-se, de forma recursiva, nas dependências do estágio gerado pelo *DAGScheduler* para realizar o processamento do *job*.

Este processo é exemplificado através do método exibido no Algoritmo 1. Primeiro, adiciona-se o RDD visitado na lista de RDDs já processados (Linha 3) e em seguida, percorre-se cada dependência adicionada na lista de dependências do RDD visitado (Linhas 4–6). Por fim, se a dependência ainda não foi processada (Linha 5), repete-se este mesmo processo para a dependência em questão (Linha 6). O resultado desta primeira etapa consiste em uma lista contendo todos os RDDs que serão criados e manipulados durante a computação do *job*.

Após a identificação dos RDDs, a próxima etapa consiste na coleta das métricas desejadas, as quais serão utilizadas no processo de gerência de memória. O processo de coleta utilizado é ilustrado pelos métodos apresentados nos Algoritmos 2, 3 e 4, cada qual responsável por obter uma determinada métrica.

O método *getLineageSize*, ilustrado pelo Algoritmo 2, visa obter o tamanho da *Lineage* de um RDD. Para isto, atualiza-se o tamanho atual do RDD, iniciando em 0 (Linha 3). Em seguida, percorre-se as dependências desse RDD (Linhas 4–9) verificando se a dependência em questão ainda não foi processada (Linha 5). Caso a dependência não tenha sido processada, adiciona-se esta à lista de nodos visitados (Linha 6) e processa-se esta nova dependência (Linha

Algoritmo 1: Identificação dos RDDs utilizados no *job*.

```
   Entrada: StageResult gerado pelo DAGScheduler.
   Saída: Lista com todos os RDDs identificados no job.
 1 rddsDependencies ← List[RDD];
 2 Function getRddsUsedInJob(rdd: RDD[_], rddsDependencies: List[RDD]) is
 3 |   rddsDependencies += rdd;
 4 |   foreach deps in rdd.dependencies do
 5 |   |   if deps ∉ rddsDependencies then
 6 |   |   |   getRddsUsedInJob(deps, rddsDependencies);
 7 |   |   end
 8 |   end
 9 end
```

Algoritmo 2: Obtenção do Tamanho da *Lineage*

```
   Entrada: List[RDD] contendo os RDDs utilizados no job
   Saída: Estrutura HashMap contendo o id do RDD e seu respectivo tamanho da
          lineage.
 1 lineageSize ← HashMap[RDD, Int];
 2 Function getLineageSize(rdd: RDD[_], rddID: Int, list: HashMap[RDD, Int],
    visited: List[RDD])
 3 |   update(lineageSize[rddId]);
 4 |   foreach deps in rdd.dependencies do
 5 |   |   if deps ∉ visited then
 6 |   |   |   visited += deps;
 7 |   |   |   getLineageSize(deps, rddID, list, visited);
 8 |   |   end
 9 |   end
10 end
```

7).

Nesta implementação é importante evitar que uma dependência seja visitada mais de uma vez, impedindo que um mesmo RDD seja contabilizado diversas vezes. O resultado desse método consiste em um mapa onde as chaves são os RDDs e o tamanho de suas *lineages*. Este método deve ser chamado de forma individual para cada RDD manipulado no *job*.

O método *getUseFrequency*, denotado no Algoritmo 3, apresenta um comportamento semelhante ao método utilizado para contabilizar o tamanho da *Lineage* (*getLineageSize)*, mas com o objetivo de obter a frequência de Acesso de um RDD. Assim, primeiro atualiza-se a frequência do RDD, iniciando em 1 (Linha 3). Na sequência, percorre-se as dependências do RDD (Linhas 4–6), processando-as.

Neste método não há necessidade de realizar o controle dos nodos visitados, uma vez que o RDD deve ser acessado para a computação de cada novo RDD gerado a partir deste. Ao

Algoritmo 3: Obtenção da Frequência de Acesso

```
   Entrada: List[RDD] contendo os RDDs utilizados no job
   Saída: Estruturas HashMap contendo o id do RDD e sua respectiva frequência de
          acesso.
1  frequencyAccess ← HashMap[RDD, Int];
2  Function getUseFrequency(rdd: RDD[_], frequencyAccess: List[RDD])
3      update(frequencyAccess[rdd]);
4      foreach deps in rdd.dependencies do
5          getUseFrequency(deps, frequencyAccess);
6      end
7  end
```

final de sua execução, o resultado é uma estrutura mapeando cada RDD com sua respectiva quantidade de acesso.

Algoritmo 4: Obtenção da Frequência de Reutilização

```
   Entrada: List[RDD] contendo os RDDs utilizados no job
   Saída: Estruturas HashMap contendo o id do RDD e sua respectiva frequência de
          reutilização.
1  reuseFrequency ← HashMap[RDD, Int];
2  Function getRDDsReuse(rddList: List[RDD], reuseFrequency: HashMap[RDD,
     Int])
3      foreach rdd in rddList do
4          dependencies ← rdd.dependencies;
5          foreach deps in dependencies do
6              update(reuseFrequency[deps]);
7          end
8      end
9  end
```

O método *getRDDsReuse*, ilustrado pelo Algoritmo 4, visa obter a frequência de reutilização de cada RDD manipulado. Esse processo é realizado de modo que, para cada RDD do *job* (Linhas 3–8), percorre-se todas as dependências do RDD (Linhas 4–7) e agrega-se a quantidade de reutilização de cada RDD em tais dependências (Linha 6). Ao final do processamento, é gerado um mapa contendo como chave o RDD e como valor a sua respectiva frequência de reutilização.

Conforme demonstrado pelo Algoritmo 5, o método responsável calcular a prioridade de cada RDD visa iterar sobre todos os RDDs identificados previamente (Linhas 5–11) e calcular sua respectiva prioridade. Este cálculo é realizado utilizando o tamanho da *lineage* (Linha 6), a frequência de acesso (Linha 7) e a frequência de reutilização (Linha 8). O resultado deste cálculo consiste na prioridade do RDD (Linha 9), a qual será mantida em uma lista contendo

todos os RDDs utilizados no *job* (Linha 10) e posteriormente ordenados de maneira decrescente por prioridade (Linha 12).

Os RDDs com pontuações maiores, encontrados se nas posições iniciais da lista, deverão ter suas partições removidas o mais tarde possível. Deste modo, prioriza-se aqueles RDDs com maior frequência de reutilização em processamentos subsequentes a fim de eliminar a necessidade de recomputá-los a cada utilização.

Algoritmo 5: Cálculo da Prioridade do RDD.

```
   Entrada: Três HashMaps contendo as métricas extraídas do job.
   Saída: Lista ordenada contendo a prioridade de cada RDD.
 1 rdds ← List[RDD];
 2 lineageSize, frequencyAccess, reuseFrequency ← HashMap[RDD, Int];
 3 Function getRddsUsedInJob()
 4 |   priority ← HashMap[RDD, Float];
 5 |   foreach rdd in rdds do
 6 |   |   size ← lineageSize[rdd];
 7 |   |   access ← frequencyAccess[rdd];
 8 |   |   reuse ← reuseFrequency[rdd];
 9 |   |   value ← reuse * (size * 0,5 + access * 0,5);
10 |   |   priority.add(rdd, value);
11 |   end
12 |   return sort(priority);
13 end
```

Após a conclusão do processamento e do cálculo das prioridades dos RDDs do *job*, o resultado final é armazenado no *Driver* da aplicação. A difusão da lista com as prioridades de cada RDD é realizada entre os nodos do *cluster* sob demanda. Isso significa que a comunicação entre um *Executor*, hospedado pelos nodos *Workers*, e o *Driver* irá ocorrer apenas quando a memória desse *Executor* estiver sobrecarregada e necessitando realizar a remoção de partições.

Nessas situações de sobrecarga da memória, o Spark *Executor* irá solicitar a lista contendo a prioridade de cada RDD do *job*. Embora haja penalização no tempo de comunicação, esta requisição é realizada utilizando comunicação síncrona entre o *Driver* e o *Executor*. O sincronismo é necessário, uma vez que a lista com as prioridades é uma informação fundamental para realizar a ordenação das partições em memória e a remoção de dados.

A requisição de atualização da lista de prioridade ocorre em duas situações: quando a lista com as prioridades está vazia e quando o identificador de uma partição não é encontrado. A ausência de um identificador na lista indica que um novo *job* iniciou e novos RDDs foram manipulados, necessitando que a lista de prioridades seja atualizada em cada *Executor*. O *status*

de lista vazia significa que o processamento da aplicação começou recentemente e nenhum bloco foi removido.

3.2.4 Análise de Correlação

A fim de verificar a correlação da Equação 3.1, proposta na Seção 3.2, realizou-se uma análise de correlação do método utilizado para calcular a prioridade de cada RDD. Esta é uma análise flexível, que pode ser usada sempre que uma variável quantitativa for estudada em função a qualquer fator de interesse (COHEN; WEST; AIKEN, 2014). Esta análise tem como objetivo medir o grau de dependência entre duas variáveis, sendo indicada para avaliar a hipótese de que o aumento ou decréscimo em uma variável está associado ao comportamento de outra.

Através do coeficiente de Pearson (BENESTY et al., 2009), responsável por representar a correlação entre as variáveis observadas, é possível medir o grau de relacionamento entre as variáveis. Este coeficiente, o qual é desprovido de unidade ou ordem de grandeza, assume valores entre -1 e 1. Valores maiores que 0 indicam uma correlação positiva, isto é, ambas as variáveis observadas movem-se na mesma direção e a relação é mais forte quando o resultado se aproxima de 1. De modo similar, valores menores que 0 indicam uma correlação negativa, ou seja, as variáveis movem-se em direções contrárias e essa correlação é mais forte quando o resultado se aproxima de -1. Por fim, um coeficiente igual a 0 indica que não há correlação entre as variáveis observadas.

Para realizar a análise de correlação da equação proposta (Equação 3.1), desenvolveu-se uma aplicação Python[5], com o objetivo de simular o comportamento e a configuração de diferentes *lineages* dentro de um *job* no Spark. Esta aplicação é composta por três funções:

a) Geração de uma *lineage*: consiste em gerar diferentes combinações de *lineages* possíveis dentro de um *job*;

b) Obtenção das métricas desejadas: visa extrair, a partir da *lineage*, as métricas referentes ao tamanho da *lineage*, à frequência de acesso e à frequência de reutilização;

c) Cálculo da prioridade de cada RDD extraído: objetiva criar uma lista de prioridades para cada combinação de tamanho da *lineage*, frequência de acesso e frequência de reutilização gerada pela aplicação.

[5] https://www.python.org/

A aplicação visa simular a extração de métricas de forma análoga ao implementado no Spark. Assim, a análise foi realizada utilizando *lineages* com tamanho variando entre 5 e 60, onde, para cada configuração de tamanho, variou-se a frequência de reutilização entre 1% e 70% do tamanho da *lineage*, totalizando 46,561 configurações para o cálculo da prioridade dos RDDs. Estas configurações de *lineage* tinham como objetivo simular o comportamento de aplicações com diferentes quantidades de reutilização de RDDs.

Os resultados desta análise demonstram que existe uma forte correlação positiva entre a prioridade calculada e a frequência de reutilização, atingindo um coeficiente de correlação superior a 0,94. Quando examinamos as demais variáveis, ambas também apresentam uma correlação positiva com a prioridade calculada, com coeficientes iguais a 0,32 e 0,013 para a frequência de acesso e tamanho da *lineage*, respectivamente.

Conforme demonstrado pela análise de correlação realizada na equação proposta, o fator determinante para o crescimento da pontuação recebida pelo RDD é a frequência de reutilização, dada a sua alta correlação com o resultado da equação proposta. As demais métricas utilizadas apresentam pouca influência no resultado final geral, exercendo maior influência em situações onde a frequência de reutilização é igual para todos os RDDs.

Uma consequência desse algoritmo baseado em prioridades é um comportamento imprevisível em aplicações onde há uma mesma frequência de reutilização de vários RDDs. Em tais situações, o critério para diferenciar uma prioridade das demais será calculada baseando-se apenas na frequência de acesso e tamanho da *lineage*. Ademais, a utilização de métricas auxiliares apresentam pouco impacto no resultado gerado, ao mesmo tempo que tornam a implementação do algoritmo proposto mais complexa.

3.2.5 Experimentos e Resultados

A validação do Algoritmo Baseado em Prioridades, apresentado na Seção 3.1, foi realizada através de experimentos em um *cluster* distribuído. Os experimentos foram conduzidos na plataforma Grid'5000, utilizando um *cluster* com 7 nodos, sendo a escolha desta configuração realizada tendo como base o tamanho do conjunto de dados utilizado nos experimentos. Assim, através desta configuração de *cluster*, torna-se possível criar cenários com diferentes cargas de acesso a memória.

Os nodos selecionados foram configurados da seguinte maneira: 1 Spark *Master*, 4 Spark *Workers* hospedando 1 *Executor* por nodo, 1 nodo executando o *HDFS Namenode* e

HDFS Datanode e 1 nodo executando o *HDFS Datanode*. Cada nodo do sistema era composto por dois Intel Xeon E5-2630 v3 @2.2GHz (8 *cores*/CPU), 128GB de memória RAM e dois HDs de 558GB, conectados via *ethernet* 10Gbps. O sistema operacional utilizado foi o Debian 8, juntamente com Java JDK 1.8.202, Spark 2.2.0 e Hadoop 2.7.1.

Para os experimentos, utilizou-se os algoritmos *PageRank*, *K-Means* e *Logistic Regression* como *benchmarks*, sendo estes implementados pelo Intel HiBench (HUANG et al., 2010). O *benchmark Logistic Regression* consiste em um tipo de análise estatística utilizada frequentemente para análise preditiva. A inclusão deste *benchmark* tem como objetivo obter uma melhor amostra de aplicações com diferentes tamanhos e disposições das *lineages*. Ainda, foram utilizados dois tamanhos de *datasets*: *small* e *large*. A Tabela 2 apresenta o tamanho destes *datasets* em cada *benchmark*.

Tabela 2 – Tamanho dos Datasets utilizados

Tamanho	PageRank	K-Means	Logistic Regression
Small	1,7 MB	574,6 MB	76,4 MB
Large	247,9 MB	3,7 GB	7,5 GB

Além da variação no tamanho dos *datasets* utilizados, cada Spark *Executor* teve a sua memória variando entre 4 configurações distintas: 1 GB, 1,5 GB, 2 GB e 4 GB. É importante ressaltar que estas configurações representam a memória total disponibilizada para o *Executor*, ou seja, incluem a Memória de Armazenamento, a Memória de Execução e a Memória do Usuário.

O espaço dedicado ao armazenamento de informações corresponde a uma fração da quantidade total de memória disponível. Deste modo, na Tabela 3 são demonstrados três valores de memória: Memória Total, Memória de Armazenamento e Memória Total de Armazenamento Disponível. A Memória Total descreve ao espaço total disponível para ser dividido entre as 3 regiões de memória do Spark (Memória de Execução, Memória de Armazenamento e Memória do Usuário). A Memória de Armazenamento se refere à fração de memória destinada ao armazenamento dos dados no *Executor*, ou seja, o espaço disponível para manter dados em *cache*. Por fim, a Memória Total de Armazenamento Disponível denota o somatório da Memória de Armazenamento de todos os *Executors* do *cluster*, simbolizando o espaço máximo para realizar o armazenamento dos dados.

As variações na quantidade de memória utilizada, bem como no tamanhos dos *datasets*, têm como objetivo avaliar O Algoritmo Baseado em Prioridades sob duas condições. A

Tabela 3 – Configuração de Memória Utilizada - Algoritmo Baseado em Prioridades

Memória Total	Memória de Armazenamento	Memória Total de Armazenamento Disponível
1 GB	366,3 MB	1465,2 MB
1.5 GB	639,3 MB	2557,2 MB
2 GB	912,3 MB	3649,2 MB
4 GB	2004,4 MB	8018,4 MB

primeira condição é em situações onde a memória disponível é suficiente para comportar o *dataset*. A segunda, em situações de sobrecarga exigindo que seja feita a substituição de partições em memória. Os resultados obtidos na experimentação representam os tempos de execução dos *benchmarks* utilizando o Algoritmo Baseado em Prioridades em contraponto ao LRU nativamente implementado pelo Spark. Assim, estes resultados são a média aritmética de 20 execuções dos *benchmarks* em cada configuração.

Os resultados demonstrados na Tabela 4 e na Figura 7 referem-se à utilização de um *dataset small*. Já a Tabela 5 e a Figura 8 exibem os resultados obtidos com um *dataset large*. Quando analisamos os resultados obtidos com o *dataset small*, durante a execução do *benchmark K-Means* houve necessidade de realizar a substituição de partições da memória. Esta necessidade pode ser verificada através dos *logs* de execução dos *benchmarks*. Nestes casos, o Algoritmo por Prioridades foi 40,74% e 23,5% mais rápido quando comparado com o LRU com as configurações de 1 GB e 1,5 GB, conforme demonstrado pela Figura 7(a). Quando há espaço disponível para todo o conjunto de dados, o desempenho de ambos os algoritmos é equivalente, uma vez que nenhuma partição foi removida da memória. Estas situações podem ser observadas nas Figuras 7(b) e 7(c), apresentando um tempo de execução similar nesta configuração.

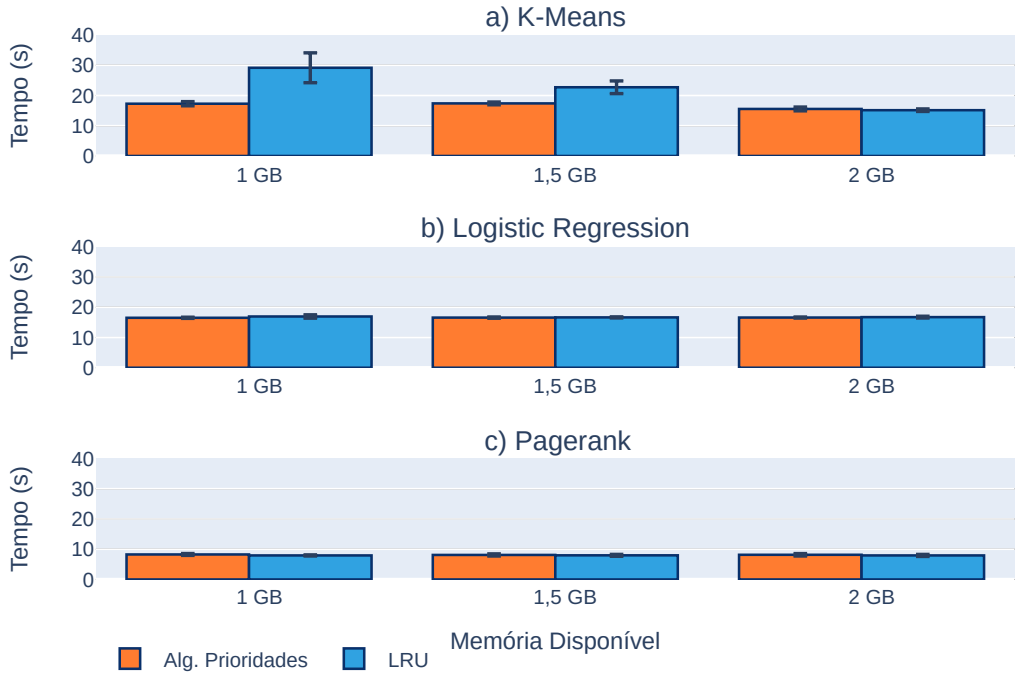
Conforme exibem a Tabela 5 e a Figura 8, quando analisamos os dados com um *dataset* maior, os resultados obtidos demonstram que o Algoritmo Baseado em Prioridades, em média, pode apresentar um desempenho igual ou superior ao LRU nas mesmas condições. Examinando os resultados do *benchmark K-Means* (Figura 8(a)), constata-se que ambos algoritmos apresentaram desempenho equivalente em todas as configurações. Uma análise dos *logs* gerados por essa aplicação revela que a memória comporta todo o *dataset*, quando utilizadas as configurações de 2 GB e 4 GB. Nas configurações de 1 GB e 1,5 GB, a memória encontra-se fortemente sobrecarregada, uma vez que o *dataset* ocupa todo o espaço disponível. Nesses casos, partições de dados serão descarregadas da memória de forma inevitável, a fim de liberar espaço para armazenamento de novos dados.

Devido a restrições de disponibilidade de memória, ambos os algoritmos não consegui-

Tabela 4 – Tempos de Execução dos Benchmarks: *Dataset small*

Benchmark	Algoritmo	Memória Disponível	Tempo de Execução (em s)	Desvio Padrão (em s)
Pagerank	LRU	1G	7,95	0,19
	Prioridades	1 GB	8,27	0,31
	LRU	1,5 GB	7,98	0,32
	Prioridades	1,5 GB	8,11	0,36
	LRU	2 GB	7,95	0,34
	Prioridades	2 GB	8,16	0,40
K-Means	LRU	1 GB	29,16	4,94
	Prioridades	1 GB	17,28	0,69
	LRU	1,5 GB	22,72	2,10
	Prioridades	1,5 GB	17,38	0,46
	LRU	2 GB	15,14	0,42
	Prioridades	2 GB	15,55	0,63
LR	LRU	1 GB	16,94	0,56
	Prioridades	1 GB	16,51	0,19
	LRU	1,5 GB	16,64	0,18
	Prioridades	1,5 GB	16,57	0,21
	LRU	2 GB	16,73	0,30
	Prioridades	2 GB	16,59	0,18

Figura 7 – Tempos de Execução dos Benchmarks: Dataset Small



ram executar o *Benchmark Logistic Regression* nas configurações com 1 GB e 1,5 GB disponíveis, conforme ilustram a Tabela 5 e a Figura 8(b). Investigando os *logs* de execução do *Logistic Regression*, verificou-se que todas as execuções foram interrompidas devido a uma exceção do tipo *OutOfMemoryError*. Esta exceção ocorre quando a JVM gasta uma grande quantidade de tempo executando a rotina de remoção de memória do *Garbage Collector*, provocado pela

Tabela 5 – Tempos de Execução do Algoritmo utilizando o *Dataset large*

Benchmark	Algoritmo	Memória Disponível	Tempo de Execução (em s)	Desvio Padrão (em s)
Pagerank	LRU	1 GB	156,40	36,92
	Prioridades		150,11	21,51
	LRU	1,5 GB	222,76	73,78
	Prioridades		194,79	59,55
	LRU	2 GB	64,90	2,46
	Prioridades		65,09	1,98
	LRU	4 GB	59,54	2,55
	Prioridades		59,69	2,53
K-Means	LRU	1 GB	454,72	11,69
	Prioridades		450,19	13,44
	LRU	1,5 GB	317,09	7,94
	Prioridades		315,76	10,28
	LRU	2 GB	149,64	6,33
	Prioridades		148,47	4,79
	LRU	4 GB	40,02	2,07
	Prioridades		39,85	1,58
LR	LRU	1 GB	0	0
	Prioridades		0	0
	LRU	1,5 GB	0	0
	Prioridades		0	0
	LRU	2 GB	1180,35	25,72
	Prioridades		1048,58	14,01
	LRU	4 GB	278,59	11,84
	Prioridades		264,68	11,09

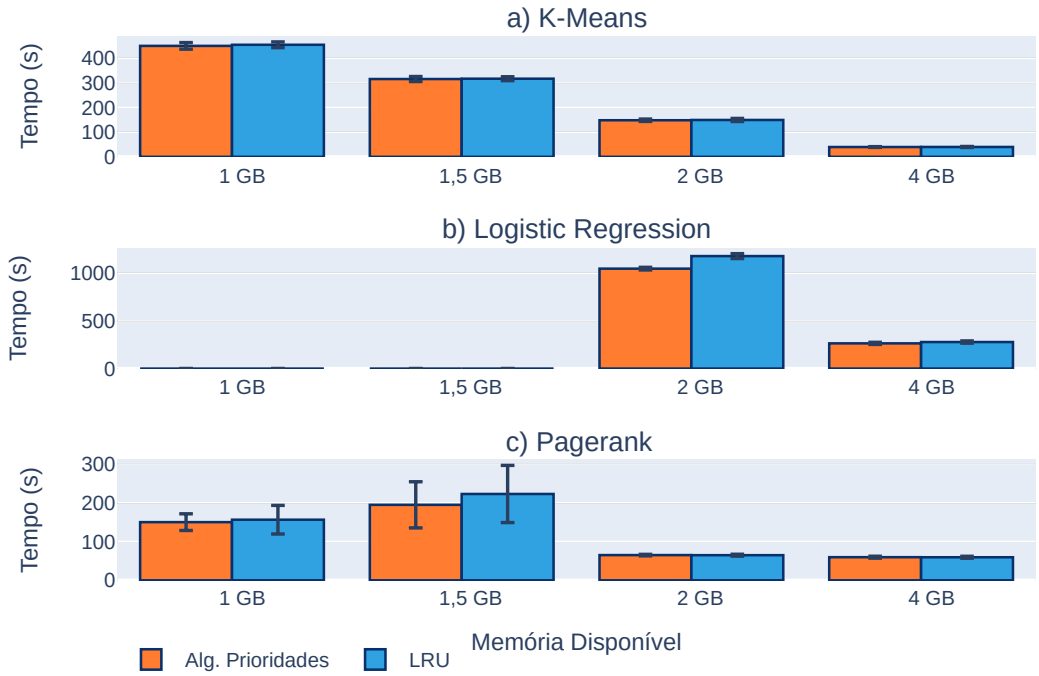
exaustão da memória disponível. Na configuração com configuração de 2 GB de memória, o Algoritmo Baseado em Prioridades reduziu em 11,16% o tempo de execução desta aplicação quando comparado ao comparado ao LRU. Por fim, utilizando 4 GB, o Algoritmo Baseado em prioridades foi 4,99% mais rápido que o LRU nas mesmas condições.

Com o *benchmark Pagerank* (Figura 8(c)), o Algoritmo por Prioridades foi 5,3% mais rápido na configuração com configuração de 1GB, e 12,56% mais rápido quando há 1,5 GB disponíveis, se comparados ao LRU nas mesmas condições. Semelhante ao ocorrido quando utilizado a configuração com *dataset* menor, nas configurações com 2 GB e 4 GB o desempenho foi equivalente, uma vez que foi possível manter todos os dados em memória e, consequentemente, não houve necessidade de substituições de partições da memória.

3.3 GERENCIAMENTO DINÂMICO DA MEMÓRIA

A utilização de métricas obtidas da aplicação em execução pode trazer bons resultados, diminuindo o tempo necessário para realizar o processamento da mesma, especialmente em

Figura 8 – Tempos de Execução dos Benchmarks: Dataset Large



cenários onde há reutilização de de dados. Entretanto, analisando o funcionamento do LRU implementado nativamente pelo Spark, percebe-se que essa solução funciona de maneira reativa. Isto significa que é executada em situações onde o Spark detecta a inexistência de espaço disponível para armazenamento de novas informações.

Na implementação do LRU nativa no Spark, após a detecção da indisponibilidade de espaço, as rotinas de gerenciamento de memória são executadas com o objetivo de remover a quantidade de dados suficientemente grande para comportar os novos dados da memória. Por consequência, a execução da aplicação é interrompida e postergada até que as rotinas de remoção de blocos da memória sejam completamente executadas.

Este trabalho implementa um modelo de Gerenciamento Dinâmico da Memória para aplicações onde há reutilização de dados. O modelo visa monitorar a aplicação em execução a fim de identificar, de forma antecipada, a necessidade de realizar a remoção de blocos de memória. Através desta abordagem, busca-se diminuir a sobrecarga das operações de substituição de blocos da memória.

A implementação do Gerenciamento Dinâmico é divida em dois componentes: um algoritmo de substituição de blocos da memória e um agente externo. O algoritmo tem como objetivo estabelecer um critério, baseando-se em informações extraídas da aplicação em execução, para decidir qual bloco deve ser removido da memória. Para tanto, este algoritmo visa

agregar a Frequência de Reutilização dos RDDs utilizados no *job* juntamente com o algoritmo LRU.

O algoritmo utilizado no Gerenciamento Dinâmico consiste em uma otimização do LRU, sendo este uma evolução do algoritmo apresentado na Seção 3.2, com o objetivo de priorizar RDDs com reutilização dentro do *job*. Para tanto, descarta-se as métricas com baixa relevância, ou seja, a Frequência de Acesso e Tamanho da *Lineage*, identificadas na análise de correlação (Seção 3.2.4), fazendo uso apenas da Frequência de Reutilização.

O algoritmo implementado no modelo de gerenciamento mantém uma lista ordenada pela Frequência de Reutilização dos RDDs identificados no *job* e, em caso de empate, utiliza-se o algoritmo LRU para realizar a sub-ordenação dessas partições. Deste modo, em situações onde não há reutilização de RDDs, o comportamento do algoritmo será semelhante ao LRU implementado nativamente pelo Spark.

O Agente Externo visa obter métricas, sendo estas o consumo de memória, a quantidade de estágios de um *job* e o *status* da execução destes estágios, através da API REST do Spark. A partir dessas métricas, busca-se antecipar a necessidade de espaço livre em memória. Assim, pode-se executar rotinas de remoção de blocos da memória em segundo plano, de modo a reduzir a sobrecarga causada por tais operações.

O modelo de Gerenciamento da Memória implementado, exibido na Figura 9, é composto por:

a) 1 nodo *Spark Master*, responsável por gerenciar os contextos das aplicações;

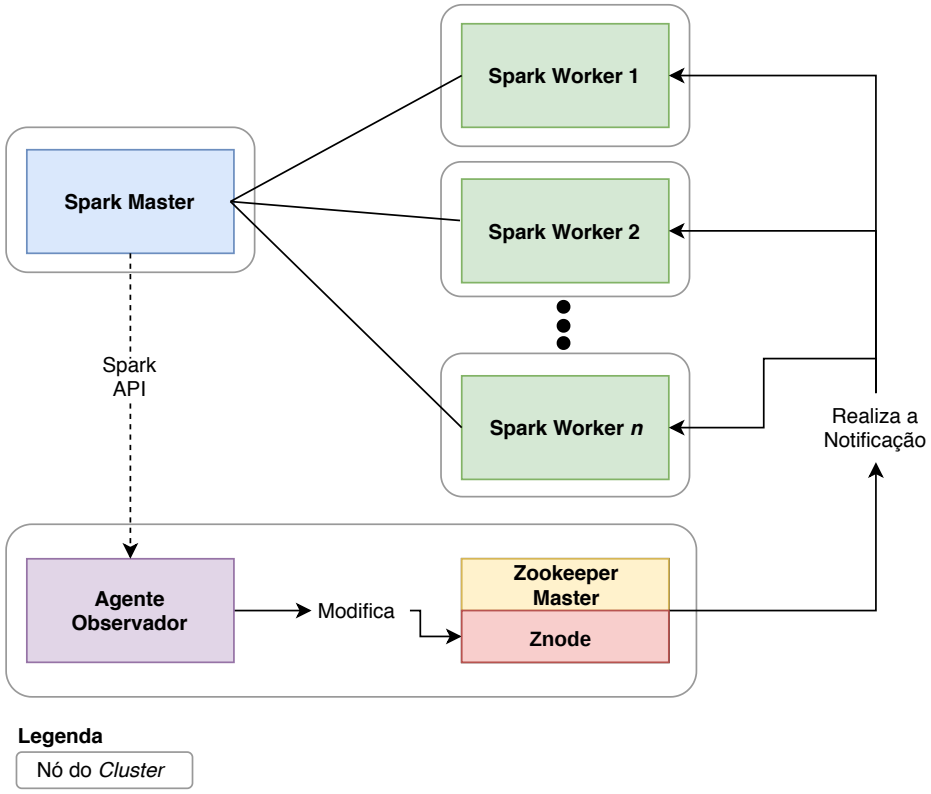
b) n nodos *Spark Workers*, os quais possibilitam a escalabilidade do *cluster* Spark;

c) o *Agente Observador*, cuja função é monitorar a aplicação;

d) 1 nodo ZooKeeper, a fim de viabilizar a comunicação entre o Agente Observador e os nodos *Spark Workers*.

No modelo de Gerenciamento Dinâmico da Memória, o *Spark Master* é nodo responsável por hospedar o *driver* da aplicação, juntamente com o seu contexto. Além disso, cabe ao *Spark Master* gerenciar e distribuir tarefas para os nodos *Workers* de modo a executar a aplicação. Durante a realização das computações requeridas pela aplicação, o *Master* disponibiliza a sua API REST, onde o usuário pode efetuar requisições com o objetivo de instrumentar e obter estatísticas relacionas à execução. Através destas requisições, pode-se identificar as aplicações

Figura 9 – Modelo de Gerenciamento Dinâmico de Memória.



Fonte: Próprio autor

executando no *cluster*, o *status* da execução destas apliações, os RDDs armazenados em *cache* e o consumo das memórias de armazenamento e execução.

O nodo ZooKeeper tem como objetivo prover os serviços deste *framework*, permtindo a sincronização e a troca de mensagens entre o *Agente Observador* e os nodos *Workers* do Spark. Deste modo, foi disposto em apenas um nodo do *cluster*, de maneira *standalone*. O nodo ZooKeeper armazena, em memória, as estruturas dos *znodes* utilizados para implementação do Gerenciamento Dinâmico. Assim, quando uma notificação a um determinado *Worker* deve ser enviada, essas estruturas são manipuladas.

O *Agente Observador* consiste em uma aplicação Java, responsável por realizar a conexão na API do Spark e efetuar o consumo dos dados, permitindo que sejam obtidas informações relativas ao estado atual das aplicações em execução. Por padrão, para cada *Job*, são coletadas informações sobre o consumo de Memória de Armazenamento e de Execução, os estágios disponíveis juntamente com seus respectivos *status* e a fração armazenada em memória do RDD em *cache*.

A partir das informações coletadas, o *Agente* busca estimar o melhor momento para

realizar a liberação de memória no Spark. Para tanto, duas métricas são adotadas: o *threshold* de ocupação de memória e o número de estágios pendentes.

O *threshold* visa definir um limite máximo para a ocupação da memória, inferior a 100%, de modo a manter uma determinada quantidade de memória livre. Assim, em situações onde novos dados devem ser armazenados em memória ou a Memória de Execução necessita de espaço extra para continuar a execução da *task*, evita-se a necessidade de interromper a execução da aplicação para realizar a remoção do espaço requerido.

Durante a execução da aplicação, caso o limite de ocupação da memória seja ultrapassado, pode haver a necessidade de remover blocos da memória. A remoção de dados é feita a fim de garantir uma fração da Memória de Armazenamento sempre disponível para a aplicação.

Uma segunda métrica utilizada consiste no *status* dos estágios criados para o processamento do *job* da aplicação. Em situações onde o *threshold* foi atingido e foram identificados estágios pendentes para serem executados, partições devem ser removidas da memória a fim de liberar espaço.

Em contraponto, se a ocupação da memória atingir o limiar e não houver estágios pendentes, nenhuma ação é tomada. Deste modo, permite-se que a memória seja completamente ocupada, uma vez que não é possível identificar de forma antecipada se a execução de um novo *job* será realizada.

Uma vez decididas quais partições devem ser removidas da memória, o Agente de Monitoramento deve realizar a notificação aos Spark *Workers* para que estes iniciem as rotinas de remoção de blocos da memória. Para tanto, o Agente calcula, de forma individual, a quantidade de memória que cada *Executor* deve liberar para ficar abaixo do *threshold* estabelecido. Na sequência, o Agente escreve esta informação no *znode* associado ao respectivo *Executor*.

3.3.1 Implementação do Gerenciamento Dinâmico

O desenvolvimento do modelo de Gerenciamento Dinâmico foi divido em três etapas de implementação: Algoritmo de gerenciamento de memória no Spark, Comunicação entre o Spark e o ZooKeeper e Agente de Monitoramento para instrumentação da aplicação.

3.3.1.1 Algoritmo de Gerenciamento de Memória

O algoritmo utilizado no modelo de Gerenciamento Dinâmico consiste em uma otimização do algoritmo LRU implementado nativamente pelo Spark. Este algoritmo une a localidade

temporal do LRU juntamente com a Frequência de Reutilização dos RDDs para definir a ordem em que as partições serão removidas da memória. Uma vez que este algoritmo necessita de informações providas pela aplicação, torna-se necessário analisar a aplicação antes de efetivamente executá-la.

A extração das informações requeridas pelo algoritmo é realizada modificando o escalonador de *jobs* do Spark, isto é, o *DAGScheduler*. Após uma ação ser aplicada ao RDD, este é submetido ao escalonador do Spark para que seu plano de execução seja gerado, possibilitando a execução do *job* no *cluster*. Entretanto, antes de iniciar a execução das *tasks* do *job*, dois métodos são executados: *getLineageGraphFromJob* e *getRddsReuseFrequency*.

O método *getLineageGraphFromJob*, exibido no Algoritmo 6, é responsável por obter o grafo de dependências do *job* a ser executado. Para isto, primeiro o RDD é adicionado na lista de nodos já computados (Linha 4), a fim de evitar que este RDD seja processado mais de uma vez. Em seguida, é gerada uma lista contendo todos os identificadores das dependências desse RDD (Linhas 5–8). Por fim, é realizado o mapeamento deste RDD junto com sua lista de dependências (Linha 9). Este procedimento é repetido para todas as dependências ainda não processadas do RDD (Linhas 10–14). Ao final deste método, a estrutura *lineage* (Linha 2) armazenará os *ids* dos RDDs juntamente com suas respectivas listas de dependências.

Algoritmo 6: Obtenção do Grafo de Dependências do *Job*.

Saída: Estrutura HashMap contendo os RDDs manipulados no *job*.

```
1  visited ← List[RDD];
2  lineage ← HashMap[Int, List[Int]];
3  Function getLineageGraphFromJob(init: RDD[_])
4      add(init, visited);
5      rddDependencies ← List[Int];
6      foreach dep in init.dependencies do
7          add(dep.id, rddDependencies);
8      end
9      add([init.id, rddDependencies], lineage);
10     foreach dep in init.dependencies do
11         if dep ∉ visited then
12             getLineageGraphFromJob(dep);
13         end
14     end
15 end
```

Após a obtenção do grafo de dependência, um segundo método é executado, visando

gerar a frequência de reutilização de cada RDD mapeado. Esse método, denominado *getRddsReuseFrequency*, é apresentado no Algoritmo 7. O objetivo do método *getRddsReuseFrequency* é acessar os RDDs identificados no *job* (Linhas 2–12), acessando a lista de dependências de cada RDD (Linha 3). Em seguida, percorre-se a lista de dependências do RDD (Linhas 4–10), contabilizando a frequência em que cada RDD aparece na lista de dependências (Linhas 5–9). Deste modo, gera-se um mapa onde a chave consiste no identificador do RDD e o valor na Frequência de Reutilização.

Algoritmo 7: Método para Obtenção da Frequência de Reutilização do RDD.

Entrada: lineage ← HashMap[Int, List[RDD]] com o grafo de dependências
Saída: reuseFrequency ← HashMap[Int, Int] com a frequência de reutilização de cada RDD

```
1  Function getRddsReuseFrequency()
2      foreach rdd in lineage do
3          dependencies ← rdd.dependencies;
4          foreach deps in dependencies do
5              if deps ∈ dependencies then
6                  updateValue(deps, reuseFrequency, +1);
7              else
8                  add(deps, reuseFrequency);
9              end
10         end
11     end
12 end
```

Uma vez realizado o processamento dos métodos *getLineageGraphFromJob* e *getRddsReuseFrequency*, o resultado gerado é mantido no *Driver*, estando disponível para os *Executors* da aplicação. O armazenamento dessas informações é feito através de um objeto do tipo *ShouldBeInCache*. A definição desta classe é ilustrada no Algoritmo 8, demonstrando seus principais métodos.

Conforme exibe o Algoritmo 8, a classe *ShouldBeInCache* é responsável pelo armazenamento das informações obtidas do *job*. O armazenamento dessas informações é realizado utilizando uma estrutura de *HashMap* contendo identificador de RDD a uma respectiva Frequência de Reutilização (Linha 2). A inserção de dados é realizada através do método *insertElement* (Linhas 3–5), a qual recebe o identificador do RDD e seu respectivo peso e adiciona ao *HashMap* (Linha 4). O método *contains* (Linhas 6–8) verifica se um RDD cujo identificador foi passada como parâmetro está armazenado no *HashMap*. Por fim, o método *getReuseFrequency* (Linhas 9–11) visa obter a Frequência de Reutilização de um RDD do RDD com identificador

Algoritmo 8: Armazenamento das Informações Processadas.

```
1  Class ShouldBeInCache:
2      rddList ← HashMap[Int, Int];
3      Function insertElemet(id: Int, score: Int)
4          add([id, score], rddList);
5      end
6      Function contains(id: Int)
7          return id ∈ rddList;
8      end
9      Function getReuseFrequency(id: Int)
10         return rddList[id];
11     end
12 end
```

informado parâmetro. Se o identificar não for encontrado, esse método retorna 1. Isso ocorre para tratar os casos onde tenta-se verificar a Frequência de Reutilização de uma partição a qual não pertence a um RDD.

A definição da classe *ShouldBeInCache* deve estender duas interfaces: *Serializable* e *Loggin*. A interface *Serializable* visa permitir a transmissão de uma instância desse objeto através de chamadas RPC (*Remote Procedure Call*), sendo este o método utilizado para realizar a comunicação entre nodos do Spark. A interface *Logging* possibilita que sejam inseridas mensagens nos *logs* de execução do Spark.

Após o processamento dos métodos implementados, a execução do *job* é realizada sem quaisquer alterações. Deste modo, o Spark divide os estágios em *tasks* e as executa nos *Executors*. Durante a realização das computações requeridas pelos estágios, a memória pode ser esgotada e blocos de dados devem ser removidos da memória a fim de completar sua execução.

A rotina de remoção de partições da memória deve priorizar a manutenção de blocos com a maior Frequência de Reutilização postergando, quando possível, a remoção destes blocos. Assim, primeiro deve-se remover blocos com menor Frequência de Reutilização e, para tanto, ordena-se os blocos mantidos em memória utilizando esta métrica.

Para realizar a ordenação dos blocos de memória é necessário obter a lista atualizada das Frequências de Reutilização de cada RDD manipulado no *job*. A fim de obter esta lista, o Spark *Executor* envia uma solicitação ao *Driver* da aplicação, requerendo a lista contendo as Frequências de Reutilização extraídas através do *DAGScheduler*. Essa solicitação é enviada sempre que, durante a rotina de remoção de dados da memória, um RDD não for encontrado na lista armazenada pelo *Executor*.

O cenário descrito indica que um novo *job* foi iniciado e, portanto, a lista com a frequência de reutilização deve ser atualizada. Durante a atualização desta lista, a execução da rotina de remoção de blocos da memória é bloqueada até que o *Executor* receba a resposta da chamada RPC realizada ao *Driver*, garantindo o sincronismo da comunicação. Este sincronismo faz-se necessário, uma vez que as informações relativas a Frequência de Reutilização dos RDDs são indispensáveis para o prosseguimento da execução da rotina de remoção de partições da memória. Como consequência dessa comunicação síncrona, a execução da *task* da aplicação é bloqueada até a conclusão da rotina de remoção de dados da memória.

Porém, a classe responsável por abrigar os métodos de gerenciamento de memória do Spark, denominada *MemoryStore*, não tem acesso ao ambiente encarregado pela comunicação entre os nodos do *cluster*. O acesso ao ambiente RPC pode ser realizado através do *BlockManager*, responsável pelo armazenamento local e remoto de blocos de dados.

Para possibilitar que os métodos de gerenciamento de memória tenham acesso ao ambiente de comunicação RPC, transmite-se uma referência do *BlockManager* ao *MemoryStore* após a sua criação, através de um método *set* criado no *MemoryStore*. Deste modo, as rotinas de gerenciamento de memória passam a ter acesso ao ambiente de comunicação e, consequentemente, torna-se possível realizar chamadas remotas diretamente destas rotinas.

Assim, quando houver a necessidade de realizar a remoção de blocos de memória, realiza-se uma chamada direta ao *Driver*, partindo do *MemoryStore*, a fim de requerer a lista de Frequência de Reutilização obtida durante a análise do *job*, visando ordenar as partições armazenadas de acordo com sua respectiva Frequência e Reutilização.

3.3.1.2 Comunicação e Notificação Utilizando o ZooKeeper

A implementação do algoritmo de gerenciamento de partições da memória garante que, em situações onde a memória contra-se sobrecarregada, as partições que a serem removidas são aquelas com a menor frequência de reutilização. Entretanto, este algoritmo funciona de maneira reativa, isto é, será executada apenas quando não houver mais espaço disponível.

A fim de implementar o modelo de Gerenciamento Dinâmico, o qual permite que partições sejam removidas antes que o espaço disponível para armazenamento de dados esteja completamente esgotado, é necessário que haja uma comunicação entre o Spark e o Agente de Monitoramento. Para tanto, após a implementação do algoritmo responsável por gerir o espaço disponível, o próximo passo consiste em implementar a comunicação entre os *Executors* do

Spark e o ZooKeeper.

A implementação desta comunicação utiliza a API Java oferecida nativamente pelo ZooKeeper, expondo os métodos necessários para criar e gerencias conexões. Entretanto, para a realizar a conexão junto ao ZooKeeper, duas informações são obrigatoriamente necessárias:

a) identificador do *Executor* para criação do seu respectivo *znode* na árvore em memória oferecida pelo ZooKeeper; e

b) endereço de IP do nodo *Master* do ZooKeeper.

O identificador do *Executor* pode ser obtido por meio do *BlockManager*, através do mesmo objeto utilizado para realizar chamadas RPC na *MemoryStore*. Este identificador é um inteiro de valor único no *cluster*, sendo utilizado para registrar o *Executor* junto ao *Driver* da aplicação para que seja possível executar *tasks* das aplicações Spark.

O endereço do IP do nodo *Master* do ZooKeeper é obtido manipulando o arquivo de configuração do Spark. Para tanto, adiciona-se a propriedade *spark.zookeeper.master.url* como chave e o IP do nodo *Master* como valor no arquivo utilizado para submeter as configurações da aplicação. Embora a propriedade seja relacionada ao ZooKeeper, é importante que esta inicie com o prefixo *spark.* uma vez que o Spark implementa filtros para a carga do arquivo de configuração, descartando e invalidando todas as configurações as quais não iniciem utilizando tal prefixo. Assim, quando o *framework* é inicializado, este carrega o arquivo de configurações de modo a permitir que todos os *Executors* do Spark passem a ter acesso às configurações realizadas durante a submissão da aplicação.

Especificamente, a conexão junto ao ZooKeeper é realizada durante a inicialização do *MemoryStore*. Neste ponto, o Spark inicializa as variáveis e métodos responsáveis pelo controle da memória consumida. A seguir, o Spark executa a rotina de conexão do *Executor* junto ao nodo *Master* do ZooKeeper, exibida no Algoritmo 9, na qual os tratamentos de erros e métodos auxiliares foram suprimidos. A partir da conexão criada por cada *Executor*, é possível monitorar estes *Executors* de forma individual e realizar a notificação para a remoção de blocos da memória da memória, quando necessário.

Conforme apresentado pelo Algoritmo 9, a conexão do cliente junto ao ZooKeeper é gerenciada pela classe *ZookeeperConnection*, implementada em Java. Esta classe recebe como parâmetros o identificador do *Executor* onde está hospedada, a referência ao objeto *MemoryStore* e a URL (*Uniform Resource Location*) do nodo *Master* do ZooKeeper.

Algoritmo 9: Gerenciamento da Conexão com o ZooKeeper.

```
   Data:
   dm ← objeto o qual implementa o método process do ZooKeeper para tratamento
   dos eventos;
   Spark_Executor_id ← identificador único do Executor do Spark;
1  Class ZookeeperConnection:
2      zk ← new ZooKeeper(Zk_IP, timeout, dm);
3      Function watchZnode()
4          myPath ← getMyPath(Spark_Executor_id);
5          zk.exists(myPath, dm);
6      end
7      Function createZnode()
8          myPath ← getMyPath(Spark_Executor_id);
9          stat ← zk.exists(myPath, dm);
10         if ∄ stat then
11             zk.create(myPath, emptyData, ZooDefs.Ids.OPEN_ACL_UNSAFE,
               CreateMode. PERSISTENT);
12         end
13     end
14 end
```

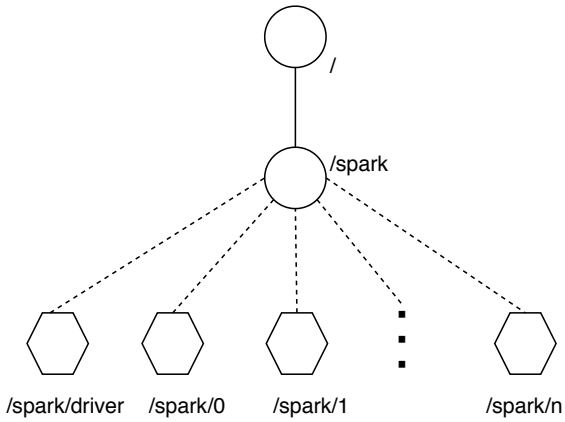
Durante a criação de um objeto desta classe, primeiro é realizada a inicialização do *DataMonitor*, cujo objetivo é tratar os eventos gerados pelo ZooKeeper. A seguir, é salvo o identificador na classe e inicializada a conexão com o ZooKeeper (Linha 2), utilizando o endereço do nodo *Master*, um *timeout* e o objeto para tratamento dos eventos, isto é, o *DataMonitor*.

Na sequência, cria-se o *znode* responsável por representar o *Executor* dentro da árvore *in memory* do ZooKeeper, utilizando a função *createZnode()* (Linhas 7–12). O caminho deste nodo é formado pelo prefixo */Spark/* seguido do seu identificador único, obtido através do *BlockManager*. Uma representação visual desta estrutura é exibida na Figura 10, apresentando a árvore de *znodes* criada após a inicialização do *cluster* Spark.

Por fim, registra-se o cliente do ZooKeeper utilizando a função *watchZnode()*(Linhas 3–6), a fim de viabilizar o recebimento de notificações de eventos ocorridos na árvore de *znodes*. Esse registro é efetuado apenas verificando a existência do nodo desejado. Após a ocorrência de um evento, o ZooKeeper irá informar todos os objetos registrados para realizar o tratamento da notificação.

Quando uma modificação ocorre em um *znode*, todos os clientes são notificados pelo ZooKeeper, sendo repassado o caminho do nodo responsável pelo disparo do evento. Cabe a cada um dos clientes verificar se o nodo é de seu interesse e, caso seja necessário, realizar a

Figura 10 – Árvore de *znodes* Criados pela Gerenciamento Dinâmico.



Fonte: Próprio autor

remoção de dados da memória. A implementação do tratamento dos eventos recebidos pelo ZooKeeper é realizada através da função demonstrada no Algoritmo 10.

O Algoritmo 10 implementa o método *process*, de modo que este seja sobrescrito a fim de implementar a lógica necessária para tratamento dos eventos disparados. Este método é exposto pela interface *Watcher* provida pelo ZooKeeper. Desse modo, o ZooKeeper consegue notificar os clientes através de chamadas ao método *process* de cada cliente conectado.

Algoritmo 10: Tratamento de Eventos Recebidos.

```
Data:
zooConnection ← Conexão com o Zookeeper;
memoryStore ← Objeto responsável pelo gerenciamento da memória do Spark;
1  WatchedEvent é a classe de evento implementada pelo Zookeeper.
2  Function process(event: WatchedEvent)
3      eventPath ← event.getPath();
4      myPath ← getMyZonePath();
5      if eventPath == myPath then
6          znodeData ← zooConnection.getData();
7          requiredBytes = Long(znodeData);
8          memoryStore.evictBlocks(requiredBytes);
9      end
10     zooConnection.watchZnode();
11 end
```

Através do método *process*, exibdo no Algoritmo 10, o *Executor* recebe a notificação de uma mudança ocorrida nos dados associados às estruturas dos *znodes*. A seguir, verifica-se em qual nodo ocorreu o evento (Linha 3). Caso o evento tenha ocorrido no *znode* de interesse do

Executor, isto é, no nodo associado ao seu identificador único (Linhas 4 – 5), deve-se remover blocos da memória. Essa remoção é realizada obtendo a quantidade de espaço que deve ser liberado, calculada e escrita pelo Agente de Monitoramento (Linhas 6–8). Ao final deve-se observar o znode de interesse, a fim de receber novas notificações relacionadas a alteração dos znodes mantidos pelo ZooKeeper (Linha 10).

3.3.1.3 Agente de Monitoramento

A última etapa para implementação do Gerenciamento Dinâmico refere-se ao desenvolvimento do Agente de Monitoramento, o qual permite a instrumentação da execução de aplicações Spark. Este agente tem como função determinar se dado o contexto da aplicação, isto é, *status* de execução da aplicação e consumo de memória dos *Executors*, deve-se ou não, remover partições da memória de forma antecipada.

O Agente de Monitoramento consiste em uma aplicação desenvolvida na linguagem Java, com o objetivo de obter dados do Spark via API REST, de modo a prever a necessidade de liberação de espaço em memória no Spark durante a execução de aplicações. O agente é construído utilizando a API oficial do ZooKeeper para realizar a manipulação dos *znodes* e, dessa forma, realizar a comunicação entre o Agente e os *Executors* do Spark.

A aplicação divide-se em três pacotes: *Models*, *Network* e *Monitor*. A Figura 11 apresenta o Diagrama de Pacotes com as definições das classes as quais compõem cada pacote.

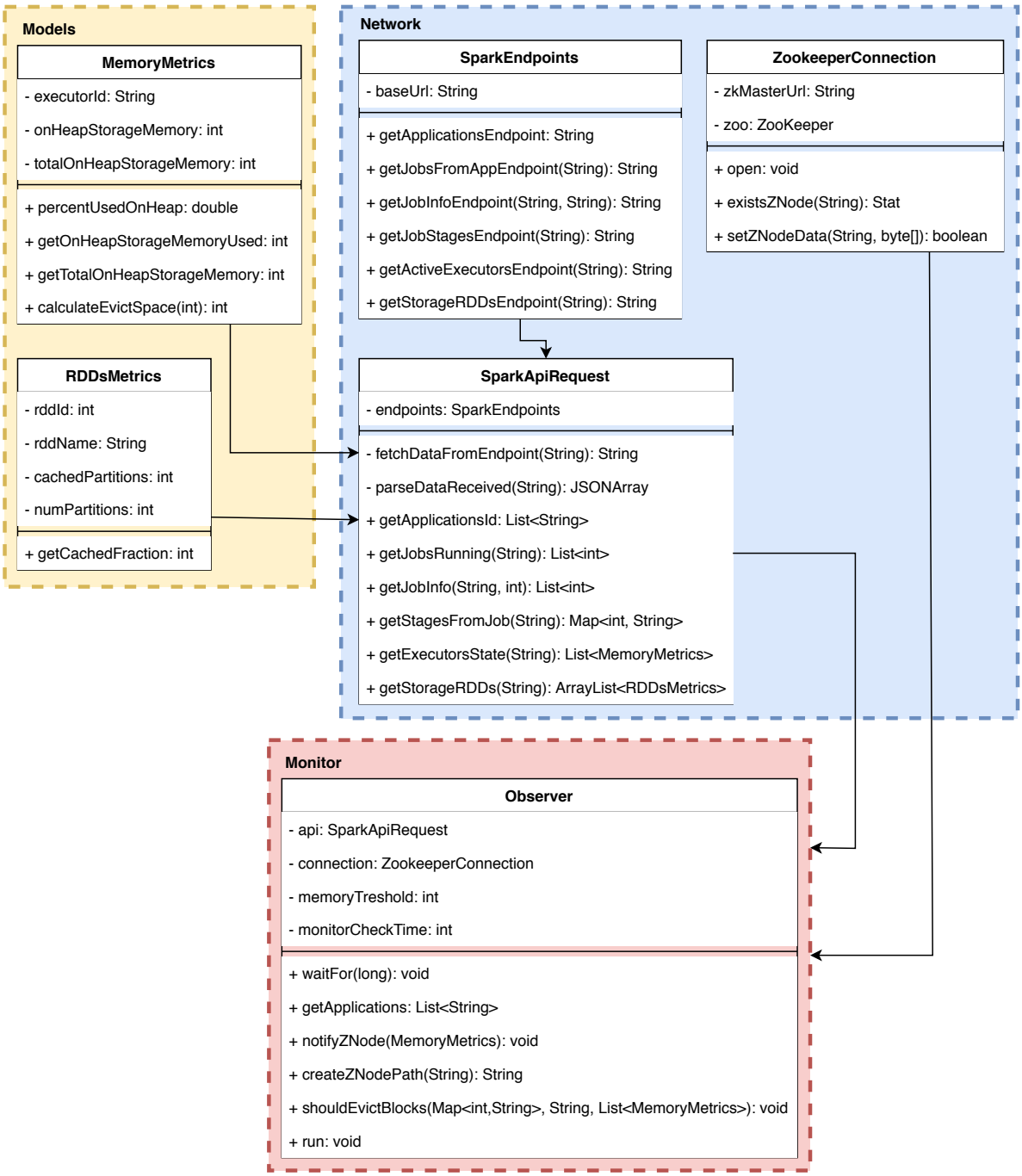
O pacote *Models* oferece classes para encapsular as principais métricas obtidas através do Spark. Para tanto, agrupa-se os dados de interesse e são oferecidos métodos para melhorar a visualização dos dados coletados. Dessa forma, torna-se mais fácil a manipulação e o tratamento destas informações. Neste pacote encontram-se duas classes: *MemoryMetrics* e *RDDsMetrics*.

A classe *MemoryMetrics* armazena dados sobre o consumo de memória dos *Executors* utilizados no processamento da aplicação. Assim, são armazenados o identificador do *Executor*, o total de Memória de Armazenamento disponível e a quantidade de Memória de Armazenamento ocupada. Além disso, nesta classe são implementados os seguintes métodos:

a) *percentUsedOnHeap*: calcula a fração, em termos percentuais, do espaço consumido da Memória de Armazenamento;

b) *getOnHeapStorageMemoryUsed*: retorna o total, em *bytes*, de Memória de Armazena-

Figura 11 – Diagrama de Pacotes do Agente de Monitoramento.



Fonte: Próprio autor

mento utilizada pela aplicação;

c) *getTotalOnHeapStorageMemory*: retorna o total, em *bytes*, de Memória de Armazenamento disponível para a aplicação;

d) *calculateEvictSpace(int)*: calula a quantidade de memória que deve ser removida do *Exe-*

cutor, baseando-se no *threshold* passado por parâmetro para o método.

Na classe *RDDsMetrics* são armazenadas as informações relativas aos RDDs mantidos em *cache*. Para tal, mantém-se o identificador do RDD, o número total de partições deste RDD e o número de partições armazenadas em memória. Além dos atributos responsáveis por armazenar as informações coletadas, esta classe implementa o método *getCachedFraction*, responsável por calcular o percentual de partições mantidas em *cache*.

O pacote *Network* concentra as classes responsáveis pelo gerenciamento das conexões realizadas pela aplicação junto ao Spark e ao ZooKeeper. O gerenciamento destas conexões é feito através das classes *SparkEndpoints*, *SparkApiRequest* e *ZookeeperConnection*.

Na classe *SparkEnpoints*, encontram-se os métodos responsáveis por abstrair os *Endpoints*, isto é, as URLs para acesso as informações do Spark. Para isto, esta classe formata as URLs corretamente com o identificador da aplicação e do *job*, quando necessário, de modo a possibilitar que sejam utilizadas para realizar as requisições na API REST do Spark.

A classe *SparkApiRequest* tem como objetivo realizar as requisições através da API REST do Spark e obter informações sobre a aplicação, sendo estas requisições realizadas utilizando os *endpoints* abstraídos pela classe *SparkEndpoints*. Todo o tratamento dessas requisições é realizado por este componente, disponibilizando métodos para obter as informações necessárias para realizar o monitoramento da aplicação. Assim, os métodos disponibilizados por esta classe são:

a) *fetchDataFromEndpoint*: acessa o *endpoint* recebido por parâmetro e retorna a resposta obtida. Retorna null em caso de falha;

b) *parseDataReceived*: decodifica o arquivo JSON recebido como resposta da requisição ao *endpoint*;

c) *getApplicationsId*: utiliza o *endpoint /application* e retorna uma lista com todas as aplicações em execução;

d) *getJobsRunning*: acessa o *endpoint /applications/[app-id]/jobs* e retorna os *jobs* em execução da aplicação cujo identificador foi recebido como parâmetro;

e) *getJobInfo*: retorna os estágios que serão utilizados para processar o *job*, obtidos através do *endpoint /applications/[app-id]/jobs/[jib-id]*;

f) *getStagesFromJob*: utiliza o *endpoint /applications/[app-id]/stages* e retorna os estágios com seus respectivos status da aplicação informada no parâmetro;

g) *getExecutorsState*: Utiliza o *endpoint /applications/[app-id]/executors* para coletar os dados relativos ao consumo de memória dos *Executors* ativos na aplicação cujo identificador foi recebido por parâmetro;

h) *getStorageRDDs*: Através do *endpoint /applications/[app-id]/storage/rdd*, obtém os RDDs mantidos em cache pela aplicação cujo identificador foi passado por parâmetro.

A última classe contida no pacote *Network* é a *ZookeeperConnection*, a qual tem como objetivo gerenciar a conexão do Agente de Monitoramento com o nodo *Master* do ZooKeeper. Os métodos oferecidos por essa classe visam abstrair a utilização da API do ZooKeeper de modo a facilitar a comunicação entre o agente e o ZooKeeper, possibilitando que seja realizada a notificação dos *Executors*. Para isto, esta classe oferece os métodos:

a) *open*: cria uma nova conexão com o nodo *Master* do ZooKeeper, permitindo acesso à árvore de *znodes*;

b) *existsZNode*: recebe como parâmetro uma string contendo o caminho do *znode* o qual deseja verificar sua existência, retornando um objeto do ZooKeeper, do tipo *Stat*, para indicar o *status* do *znode*. desse *znode*;

c) *setZNodeData*: recebe como parâmetro uma string com o caminho do *znode* e um *array* de *bytes* com os dados que devem ser escritos no *znode*. Retorna *true* caso a operação tenha sido realizada com sucesso e *false* em caso de falha ou inexistência do *znode*.

No pacote *Monitor* encontra-se a classe *Observer*, a qual contém a lógica principal para o monitoramento da aplicação. Para tanto, esta classe obtém as informações relacionadas à aplicação utilizando as classes implementadas nos pacotes *Models* e *Network*. A partir das informações, o Agente de Monitoramento decide se partições de RDDs devem ser removidas da memória. Para tanto, nesta classe são implementados os métodos:

a) *waitFor*: interrompe a execução da *thread* por um determinado tempo, definido pelo parâmetro do método;

b) *getApplications*: obtém as aplicações em execução. Esta função é executada até que seja encontrada, pelo menos, uma aplicação ativa no Spark;

c) *notifyZNode*: calcula o espaço de memória que deve ser liberado e notifica o *Executor* recebido por parâmetro;

d) *createZNodePath*: recebe uma string com o identificador do *Executor* e retorna seu caminho na árvore de *znodes*;

e) *shouldEvictBlocks*: método responsável por determinar se partições de RDDs devem ser removidas da memória, baseando-se nas características do ambiente atual de execução;

f) *run*: função que executa a *thread* principal de monitoramento do agente.

A implementação do monitoramento é realizada em uma *thread* separada, independente da *thread* principal. Assim, é possível enviar comandos para o Agente de Monitoramento e, deste modo, acompanhar o percentual dos RDDs mantidos em *cache*. Uma visão geral do fluxo de execução da *thread* de monitoramento é ilustrada pelo Algoritmo 11.

Algoritmo 11: *Thread* de Monitoramento da Aplicação.

```
   Data:
   api ← Conexão com o Zookeeper;
   monitorCheckTime ← Periodicidade do monitoramento;
 1 Function run()
 2     api ← openZookeeperConnection();
 3     while True do
 4         apps ← getApplications();
 5         while True do
 6             jobsRunning ← api.getJobsRunning();
 7             stageStatus ← api.getStageStatus(jobsRunning);
 8             memoryMetrics ← api.getMemoryMetrics(jobsRunning);
 9             rddsCached ← api.getRddsCached(jobsRunning);
10             if ocorreu erro na obtenção dos dados da api then
11                 break;
12             end
13             shouldEvictBlocks(stageStatus, memoryMetrics);
14             waitFor(monitorCheckTime);
15         end
16     end
17 end
```

De acordo com o Algoritmo 11, para implementar a *thread* de monitoramento, primeiro é inicializada a conexão entre o Agente de Monitoramento e o ZooKeeper (Linha 2). Um vez estabelecida a conexão, o monitoramento é realizado utilizando dois laços aninhados. O laço

externo (Linhas 3–16) obtém as aplicações em execução no Spark através do método *getApplications* (Linha 4). Este método é implementado para que seja possível obter os identificadores das aplicações em execução. Assim, garante-se a existência do identificador da aplicação para futuras requisições.

No laço interno (Linhas 5–15) são executadas as rotinas para coleta das métricas da aplicação implementadas pela classe *SparkApiRequest* do pacote *Network*. Primeiro, o agente obtém os *jobs* da aplicação em execução (Linha 6). A seguir, coleta-se os estágios deste *job* (Linha 7), juntamente com seus respectivos *status*. Um estágio pode ter os seguintes *status*: *pending*, quando encontra-se pendente para execução; *running* quando está em execução e *completed* quando sua execução já foi realizada.

Na sequência, são reunidas as informações sobre os *Executors* ativos na execução da aplicação (Linha 8). Nesta etapa, são extraídos os identificadores dos *Executors* e as informações relativas ao consumo de memória. O identificador obtido nesta requisição é o mesmo utilizado para criar seu respectivo *znode* no Zookeper. Desta forma, é possível notificar individualmente cada *Executor*. A última requisição consiste na coleta das informações dos RDDs mantidos em *cache* pela aplicação (Linha 9). A partir desta informação é possível determinar quais RDDs estão sendo mantidos em memória pela aplicação.

Após a execução de cada método utilizado para realizar o monitoramento (Linhas 6, 7, 8 e 9), deve-se verificar se o retorno dos método executado é igual a *null*. Esta verificação é necessária para garantir que os dados foram corretamente obtidos (Linhas 10–12). Caso o retorno seja *null*, a aplicação Spark finalizou sua execução durante o processamento do *loop* interno (Linhas 5–15) e este deve ser finalizado através do comando *break*. Na sequência, o laço externo (Linhas 3–16) volta a ser executado, obtendo novamente as aplicações em execução, já que o identificador da aplicação mudou pois trata-se de uma nova aplicação em execução.

Uma vez coletadas as métricas, o próximo passo consiste em verificar se há a necessidade de remover blocos da memória. Para isso, o método *shouldEvictBlocks* (Linha 13) verifica os estágios do *job*, bem como o consumo de memória dos *Executors* da aplicação. Caso haja estágios pendentes para execução e o percentual de ocupação da memória esteja acima de um *threshold* pré-estabelecido, uma notificação é enviada ao *Executor* indicando que este deve remover partições de RDDs da memória. O *threshold* é um limiar máximo para ocupação da memória utilizada pelo Spark, com valor inferior a 100%. Por padrão, o Agente de Monitoramento adota um limiar de ocupação máximo de 95%. Entretanto, o usuário pode configurar

este valor no momento da inicialização do agente.

Para realizar a notificação do *Executor* cuja ocupação da memória excedeu o *threshold* máximo, o Agente de Monitoramento calcula a quantidade de memória, em *bytes*, acima do *threshold* e escreve este valor no *znode* correspondente ao *Executor* modificando-o. Esta alteração é detectada pelo ZooKeeper, o qual envia uma notificação para todos os clientes conectados. O *Executor* recebe esta informação contendo o espaço de memória que deve ser liberado e executa a remoção dos blocos de memória. A remoção de partições é realizada até que o espaço liberado pelo *Executor* seja maior ou igual ao requisitado pelo Agente de Monitoramento.

Uma iteração do laço interno (Linhas 5–15) é finalizada aguardando um tempo pré-estabelecido, até que a próxima iteração deste mesmo laço execute e toda a computação seja realizada novamente. Por padrão, realiza-se uma iteração por segundo, de modo a acompanhar constantemente a evolução da ocupação da memória. Intervalos de tempo superiores tendem a deixar a memória sobrecarregada, induzindo o Spark a gerenciar a memória de forma reativa. Assim como o *threshold*, a periodicidade com que o laço será executado pode alterada pelo usuário através dos parâmetros de entrada da aplicação, via linha de comando.

3.3.2 Experimentos e Resultados

O processo de validação do modelo de Gerenciamento Dinâmico implementado foi realizado na plataforma Grid'5000, utilizando um *cluster* com 8 nodos. Assim como os experimentos realizados utilizando o Algoritmo Baseando em Prioridades, a experimentação do modelo de Gerenciamento Dinâmico foi feita utilizando 7 nodos para o Spark, gerando diferentes cargas de acesso a memória. Entretanto, adicionou-se um 8º nodo ao *clustes* a fim de hospedar o Agente de Monitoramento e o ZooKeeper. A adição deste nodo visa evitar que haja uma sobrecarga nos nodos utilizados pelo Spark, isolando as computações requeridas por estas aplicações em um nodo dedicado.

Assim, os 8 nodos selecionados foram configurados da seguinte maneira: 1 Spark *Master*, 4 Spark *Workers*, 1 nodo executando o *HDFS Namenode* e *HDFS Datanode*, 1 nodo executando o *HDFS Datanode* e 1 nodo hospedando o Agente de Monitoramento e o ZooKeeper *Master*. Cada nodo do sistema era composto por dois Intel Xeon E5-2630 v3 @2.4GHz (8 *cores*/CPU), 128GB de memória RAM e dois HDs 600GB, conectados via quatro *ethernet* 10Gbps. O sistema operacional utilizado foi o Debian 8, juntamente com Java JDK 1.8.202, Spark 2.2.0 e Hadoop 2.7.1.

Os experimentos foram realizados utilizando os algoritmos *PageRank*, *K-Means* e *Logistic Regression* como *benchmarks*, com o *dateset large* como conjunto de dados de entrada. Tanto os *benchmarks* quanto o *dataset* utilizados na validação do modelo de gerenciamento foram implementados pelo Intel HiBench (HUANG et al., 2010). Através destes *benchmarks*, visava-se criar 3 possíveis cenários com reutilização de dados:

a) *PageRank*: acesso intensivo à memória, mas sem substituição de dados do *benchmark*;

b) *K-Means*: adição e remoção de RDDs mantidos em *cache* durante a execução do *benchmark*, além da remoção de dados da memória e;

c) *Logistic Regression*: ocorrência de remoção de dados durante o processamento do *benchmark*, mas sem realizar a remoção de RDDs da *cache* ao longo de toda execução.

Ainda, para cada *benchmark* foram criados cinco configurações de memória total disponível: 1 GB, 1,5 GB, 2 GB, 2,5 GB e 3 GB. Diferentemente dos experimentos realizados na Seção 3.2, na validação do modelo de gerenciamento optou-se por remover a configuração de 4 GB disponíveis, uma vez que este não causa sobrecarga à memória, tornando indiferente o algoritmo de gerenciamento de memória utilizado.

Por fim, foram incluídas as configurações de 2,5 GB e 3 GB de memória disponível. A inclusão destas configurações teve como objetivo criar cenários com intervalos regulares de 0,5 GB a fim de verificar o comportamento da execução dos *benchmarks* utilizados. Na Tabela 6, são demonstrados os valores de Memória Total, Memória de Armazenamento e Memória de Armazenamento Total obtidos a partir das cinco configurações utilizadas.

Tabela 6 – Configuração de Memória Utilizada - Gerenciamento Dinâmico

Memória Total	Memória de Armazenamento	Memória Total de Armazenamento Disponível
1 GB	366,3 MB	1465,2 MB
1,5 GB	639,3 MB	2557,2 MB
2 GB	912,3 MB	3649,2 MB
2,5 GB	1185,6 MB	4742,4 MB
3 GB	1458,6 MB	5834,4 MB

Para a realização dos experimentos, as únicas configurações exigidas pelo ZooKeeper consistem na porta utilizada pelos clientes para realizar a conexão – sendo 2181 a porta padrão –, um diretório para armazenamento de *snapshots* do estado da memória e um *tick time*, responsável por definir a periodicidade de mensagens *heartbeat* enviadas pelo *framework* a fim de verificar se a conexão dos clientes ainda encontra-se ativa.

O Agente de Monitoramento foi configurado para realizar a verificação do consumo de memória dos *Executors* de forma constante, sendo esta realizada uma vez por segundo. Esta janela de tempo foi escolhida visto que as *tasks* podem ser rapidamente executadas e o espaço disponível na memória ser completamente ocupado.

O *threshold* de ocupação da memória foi adotado em 95%, visando deixar 5% da Memória de Armazenamento livre em situações onde é identificada a necessidade de novas computações. A escolha desse *threshold* foi feita de maneira experimental, isto é, após testes preliminares utilizando quatro configurações de *thresholds*: 80%, 85%, 90% e 95%. A partir destes experimentos preliminares, foi possível perceber que o limiar de ocupação de 95% apresentou o melhor desempenho.

Na prática, utilizar um *threshold* baixo implica em limitar o espaço disponível para armazenamento de informações. Por exemplo, em uma configuração onde há 2,5 GB de memória total e utiliza-se um *threshold* de 70%, o espaço disponível para armazenamento de informações é equivalente a 638,61 MB, sendo este valor equivalente a configuração de 2 GB onde há 639,3MB disponíveis.

Deste modo, os valores utilizados para as métricas de Frequência de Monitoramento e *threshold* foram escolhidos após a realização de experimentos com configurações distintas. Estes experimentos visavam variar a frequência do monitoramento, isto é, o tempo entre cada verificação do consumo de memória, e o *threshold* utilizados. Assim, os valores dos parâmetros adotados na experimentação final foram aqueles que apresentaram resultados mais promissores dentre as configurações avaliadas. Por fim, os resultados apresentados são obtidos a partir da média aritmética de 20 execuções para cada configuração.

3.3.2.1 Pagerank

O *PageRank* é um *benchmark* cujo objetivo é classificar *links* baseando-se em suas ligações, visando determinar uma relevância para cada *link*, onde a precisão dessa classificação depende do número de iterações. Este *benchmark* tinha como objetivo gerar um cenário com acesso intensivo à memória, mas sem substituição de dados do *benchmark*.

No Spark, o processamento deste *benchmark* é realizado em apenas um único *job*. Ou seja, os RDDs responsáveis por calcular a relevância de cada *link* são criados e, ao final, utiliza-se uma ação para submeter a *lineage* ao *DAGScheduler* e iniciar o processamento do *benchmark*.

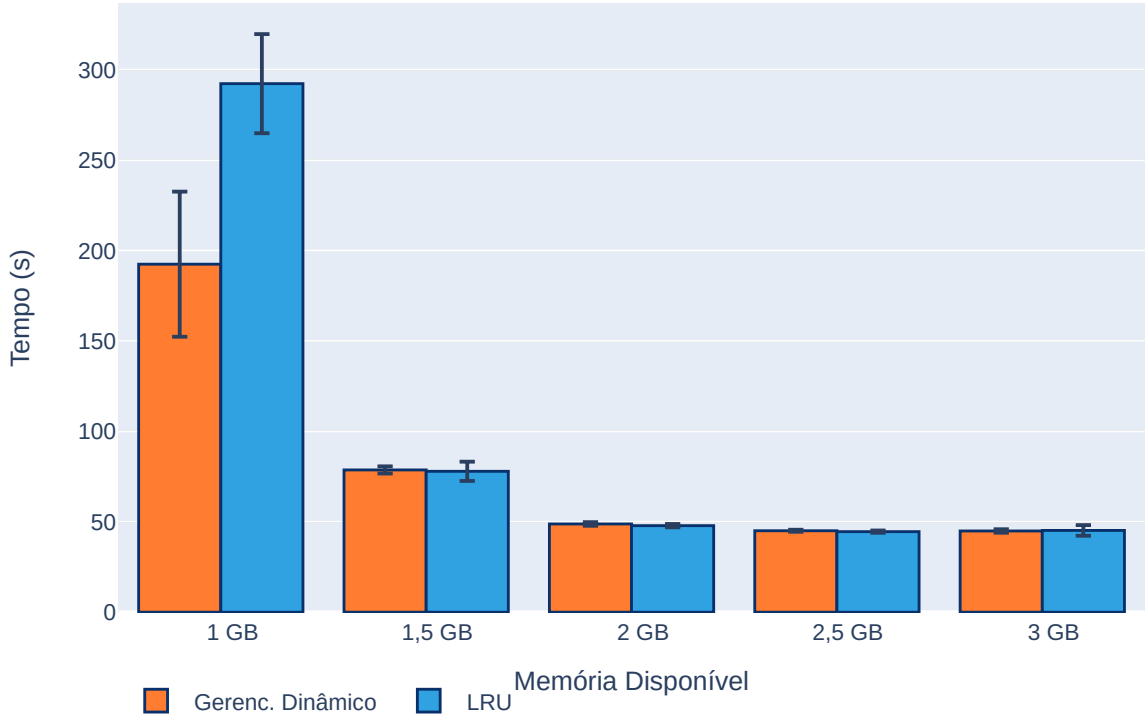
A *lineage* gerada pelo *benchmark* é composta por 30 RDDs, onde há reutilização do

RDD com identificador 6 em outros 4 novos RDDs. Estes RDDs utilizados pelo *benchmark* são processados pelo Spark em 6 estágios de execução. Assim, os resultados obtidos são demonstrados pela Tabela 7 e pela Figura 12, sendo estes resultados a média aritmética de 20 execuções.

Tabela 7 – Tempos de Execuções Obtidos no *Benchmark PageRank*

Algoritmo	Memória Disponível	Tempo de Execução (em s)	Desvio Padrão (em s)
LRU	1 GB	292,28	27,41
Gerenc. Dinâmico		192,46	40,12
LRU	1,5 GB	77,92	5,33
Gerenc. Dinâmico		78,68	1,99
LRU	2 GB	47,87	0,93
Gerenc. Dinâmico		48,81	1,01
LRU	2,5 GB	44,56	0,64
Gerenc. Dinâmico		45,05	0,57
LRU	3 GB	45,23	2,94
Gerenc. Dinâmico		44,92	0,97

Figura 12 – Tempos de Execução do *benchmark PageRank*



Fonte: Próprio autor

A diferença mais notável foi detectada na configuração com 1 GB de memória disponível. Nesta configuração, o Gerenciamento Dinâmico foi, em média, 34,15% mais rápido que o

algoritmo LRU nas mesmas condições. Analisando os *logs* de execução produzidos durante a execução deste *benchmark*, é possível perceber que não houve remoção de partições de RDDs da memória em ambos os casos, tanto com a utilização do algoritmo LRU quanto o Gerenciamento Dinâmico. Isto se dá mesmo em casos de restrição de memória total, como é possível perceber no uso de 1 GB.

Na execução deste *benchmark*, apenas variáveis de *broadcasts*, as quais são utilizadas para sincronizar a execução entre os nodos do *cluster*, são removidas da memória. Neste processo, é possível perceber que o LRU introduz de forma mais frequente um *overhead* na remoção, pela JVM, dos objetos não mais utilizados, sendo este fato demonstrado pelo exceção *GC Overhead Limit Exceeded error*.

A sobrecarga na remoção de objetos pela JVM é causada pela forma de implementação do LRU no Spark. O Spark utiliza-se de uma lista dinâmica, inicialmente com 32 posições, sendo novas posições alocadas quando a lista atinge 75% de ocupação. Assim, a remoção de blocos na memória de execução e na memória de dados sobrecarrega o *Garbage Collector*.

Diferentemente do LRU, a implementação do modelo de Gerenciamento Dinâmico utiliza uma estrutura onde não há alocação de memória realizada de maneira prévia. Desde modo, aloca-se espaço na memória apenas quando há necessidade de armazenar novos dados. Como consequência, o Gerenciamento Dinâmico consegue atingir uma mais estabilidade na execução deste *benchmark* em situações onde há uma alta sobrecarga no acesso a memória e o espaço disponível total é reduzido, acarretando em uma diminuição do tempo médio de execução.

3.3.2.2 K-Means

O *benchmark K-Means* é uma aplicação de aprendizagem de máquina de forma não supervisionada, cujo objetivo é agrupar o conjunto de dados utilizados como entrada em K grupos distintos. Este agrupamento é realizado utilizando as características encontradas no conjunto de dados. Uma particularidade deste *benchmark* é a manipulação dos RRDs mantidos em *cache*, onde diferentes RDDs são reutilizados na criação de novos RDDs.

No *benchmark K-Means* é esperado que RDDs sejam adicionados e removidos da *cache*. A *lineage* formada pela implementação desse *benchmark* é composta por 31 RDDs, dos quais 4 são mantidos em *cache*, sendo estes os RDDs 2, 3, 9 e 13. Os RDDs 9 e 13 são adicionados e posteriormente removidos da *cache* no decorrer da execução da aplicação. Os RDDs 2 e 3 apresentam as maiores frequências de reutilização, sendo mantidos em *cache* durante toda a

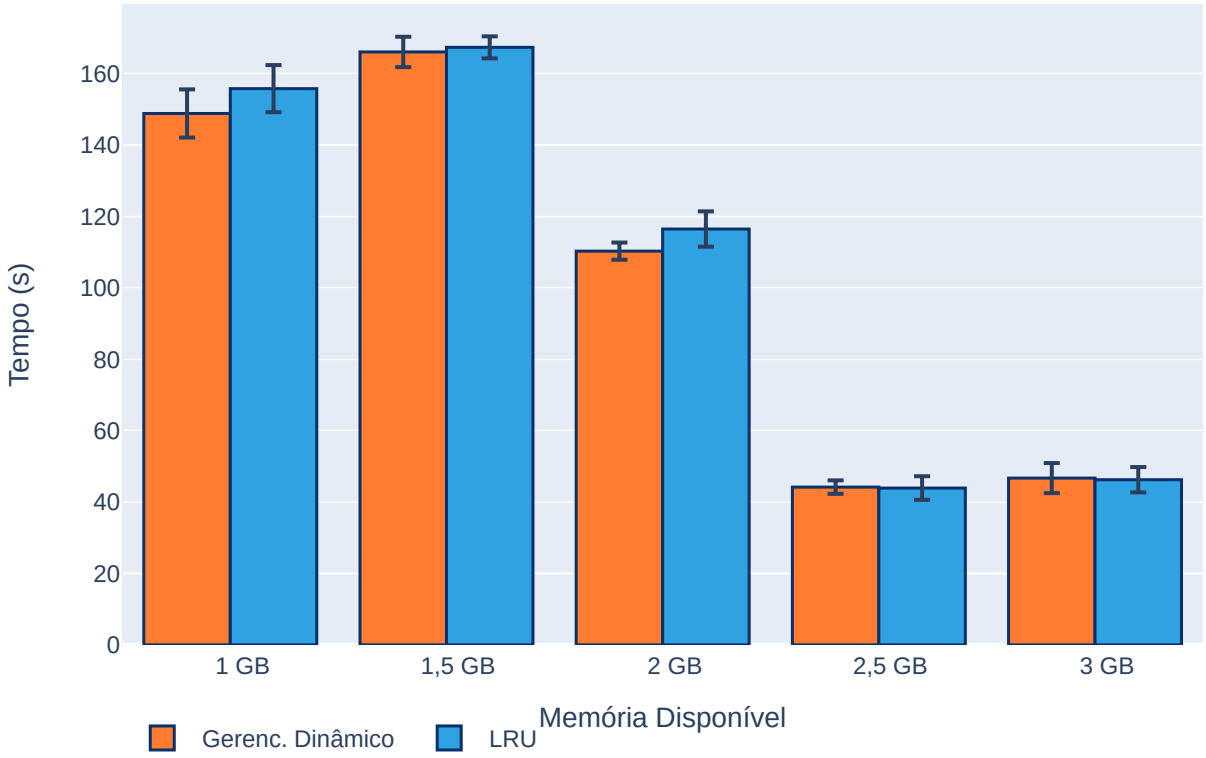
execução do *benchmark*.

Os resultados, obtidos a partir da média aritmética de 20 execuções, deste *benchmark* são demonstrados na Tabela 8 e na Figura 13.

Tabela 8 – Tempos de Execuções Obtidos no *Benchmark K-Means*

Algoritmo	Memória Disponível	Tempo de Execução (em s)	Desvio Padrão (em s)
LRU	1 GB	155,79	6,60
Gerenc. Dinâmico		148,84	6,75
LRU	1,5 GB	167,36	3,08
Gerenc. Dinâmico		166,07	4,25
LRU	2 GB	116,47	4,95
Gerenc. Dinâmico		108,28	2,41
LRU	2,5 GB	43,93	3,31
Gerenc. Dinâmico		44,19	1,89
LRU	3 GB	46,26	3,55
Gerenc. Dinâmico		46,72	4,21

Figura 13 – Tempos de Execução do *benchmark K-Means*.



Fonte: Próprio autor

Analisando os resultados obtidos neste *benchmark*, as diferenças mais perceptíveis foram detectadas nas configurações de 1 GB e 2 GB de memória total disponível. Com 1 GB, o

modelo de Gerenciamento Dinâmico foi aproximadamente 4,46% mais rápido, quando comparado ao LRU. Já na configuração de 2 GB de memória, o modelo de gerenciamento foi 7,89% mais rápido que o LRU nas mesmas condições, quando consideramos a média dos tempos de execução destes *benchmarks*.

Nas configurações de 1,5 GB, 2,5 GB e 3 GB de memória disponível, o desempenho de ambos algoritmos foi equivalente. Analisando os *logs* de execução gerados pelo *K-Means*, observou-se que a execução deste *benchmark* se deu em 14 *jobs*, sendo 7 *jobs* compostos por um estágio e os demais compostos por 2 ou 3 estágios. Em casos onde o *job* é composto por apenas um estágio, não há possibilidade de prever a existência de computações futuras. Esta restrição ocorre uma vez que o Spark executa as aplicações gerando o plano de execução do *jobs* apenas quando ações são aplicadas ao RDD.

Assim, nos *jobs* com um estágio é inviável analisar a aplicação de forma integral a fim de verificar se outros *jobs* serão necessários para completar execução. Consequentemente, o Agente de Monitoramento opta por não remover dados da memória, mesmo em situações onde o *threshold* foi ultrapassado.

Uma diferença evidente entre o algoritmo LRU e o modelo de Gerenciamento Dinâmico é a forma como os blocos de dados são operados. Na Tabela 9 é exibido o número de inserções e remoções realizadas na memória em cada configuração testada. A partir desta tabela, busca-se investigar e apresentar o comportamento das operações ocorridas em memória, quando utilizado o LRU e o algoritmo de Gerenciamento Dinâmico.

A tabela descreve o número de partições de RDDs adicionadas e removidas da memória e a quantidade de variáveis de *broadcast* adicionadas e removidas. As partições são blocos que contém os dados em *cache* da aplicação, enquanto as variáveis de *broadcast* são utilizadas pelo Spark para distribuir as *tasks* entre os *Executors*. Os valores apresentados são a média aritmética das 20 execuções realizadas em cada configuração de memória.

De acordo com a Tabela 9, com 1 GB de memória, tanto o LRU quanto o Gerenciamento Dinâmico removeram e inseriram uma quantidade semelhante de partições de RDDs e variáveis de *broadcast* durante a execução do *benchmark*. Esta semelhança na quantidade de remoções e inserções de dados na memória demonstra que, nessas condições, a memória encontrava-se fortemente sobrecarregada. Nas configurações de 1,5 GB, a diferença principal foi na quantidade de variáveis de *broadcast* removidas pelo modelo de gerenciamento. Nesta configuração, o Gerenciamento Dinâmico removeu mais variáveis de *broadcast*, e consequentemente manteve

Tabela 9 – Número de Remoções e Inserções de Dados na Memória do *benchmark* K-Means

Algoritmo	Memória	Partições de RDDs Adicionadas	Partições de RDDs Removidas	Variáveis de Broadcast Adicionadas	Variáveis de Broadcast Removidas
LRU	1 GB	185	108	147	119
Gerenc. Dinâmico		188	109	147	115
LRU	1,5 GB	356	265	146	63
Gerenc. Dinâmico		362	270	146	115
LRU	2 GB	247	150	146	65
Gerenc. Dinâmico		210	116	146	111
LRU	2,5 GB	132	12	146	63
Gerenc. Dinâmico		134	14	146	60
LRU	3 GB	133	14	146	57
Gerenc. Dinâmico		145	25	146	60

mais partições de RDDs em memória.

Na configuração de 2 GB de memória é observada a maior diferença entre os métodos de gerenciamento de memória. Nesta configuração, o Gerenciamento Dinâmico removeu uma quantidade menor de partições de RDDs da memória, havendo também uma redução no número de inserções de partições. Estas reduções ocorrem porque o algoritmo utilizado para gerenciar a memória no modelo de Gerenciamento Dinâmico consegue priorizar partições de RDDs em detrimento das variáveis de *broadcast*. Assim, enquanto o LRU remove uma partição de RDD para inserir outra partição de RDD, o algoritmo utilizado o Gerenciamento Dinâmico remove primeiro as variáveis de *broadcast*, para então remover dados.

Quando utilizadas as configurações de 2,5 GB e 3G GB de memória, o comportamento de ambos algoritmos foi similar, ou seja, tanto o tempo de execução quanto o número de operações de adição e remoção de blocos da memória foram semelhantes. Esta semelhança se dá devido ao o espaço disponibilizado se mostrou próximo do suficiente para comportar todo o *dataset*, exigindo poucas operações de adição e remoção de dados da memória.

Portanto, ao sumarizar as informações das Tabelas 8 e 9, o algoritmo utilizado no Gerenciamento Dinâmico tende a priorizar a manutenção de partições de RDDs em detrimento das variáveis de *broadcast*. Deste modo, esta característica na ordem de remoção dos dados difere entre o LRU e o algoritmo utilizado no Gerenciamento Dinâmico. Assim, em situações onde há restrição no espaço de memória disponível, como o ocorrido nos casos com 1,5 GB e 2 GB de memória total, onde o Gerenciamento Dinâmico removeu uma maior quantidade de variáveis de *broadcast*. Além disto, o Gerenciamento dinâmico conseguiu reduzir o tempo de execução

necessário nas configurações de 1 GB, 1,5 GB e 2 GB.

3.3.2.3 Logistic Regression

O *benchmark Logistic Regression* consiste em um tipo de análise estatística utilizada frequentemente para análise preditiva. Diferente dos *benchmarks* anteriores, a principal característica do *Logistic Regression* está na permanência de RDDs em memória. Assim, dois RDDs são computados e mantidos em *cache* durante toda a execução deste *benchmark*. Dessa forma, não há remoção de RDDs da *cache* como ocorrido no *K-Means*.

Durante a computação desse *benchmark*, ao total são manipulados 157 RDDs, sendo os RDDs 2 e 11 mantidos em *cache* pernamentemente durante a execução do *benchmark*. Estes RDDs manipulados são processados em 42 *jobs*, dentre os quais apenas 2 são compostos por um único estágio. Assim, torna-se possível identificar se novas computações serão requeridas pelo *job* e, se necessário, realizar a remoção antecipada de blocos da memória.

Além disso, o *benchmark Logistic Regression* possui o maior *dataset* dentre os *benchmarks* utilizados, com um total de 7,5 GB de dados. Assim, devido ao grande número de RDDs processados e ao conjunto de dados utilizado, este *benchmark* utiliza a memória de maneira mais intensiva.

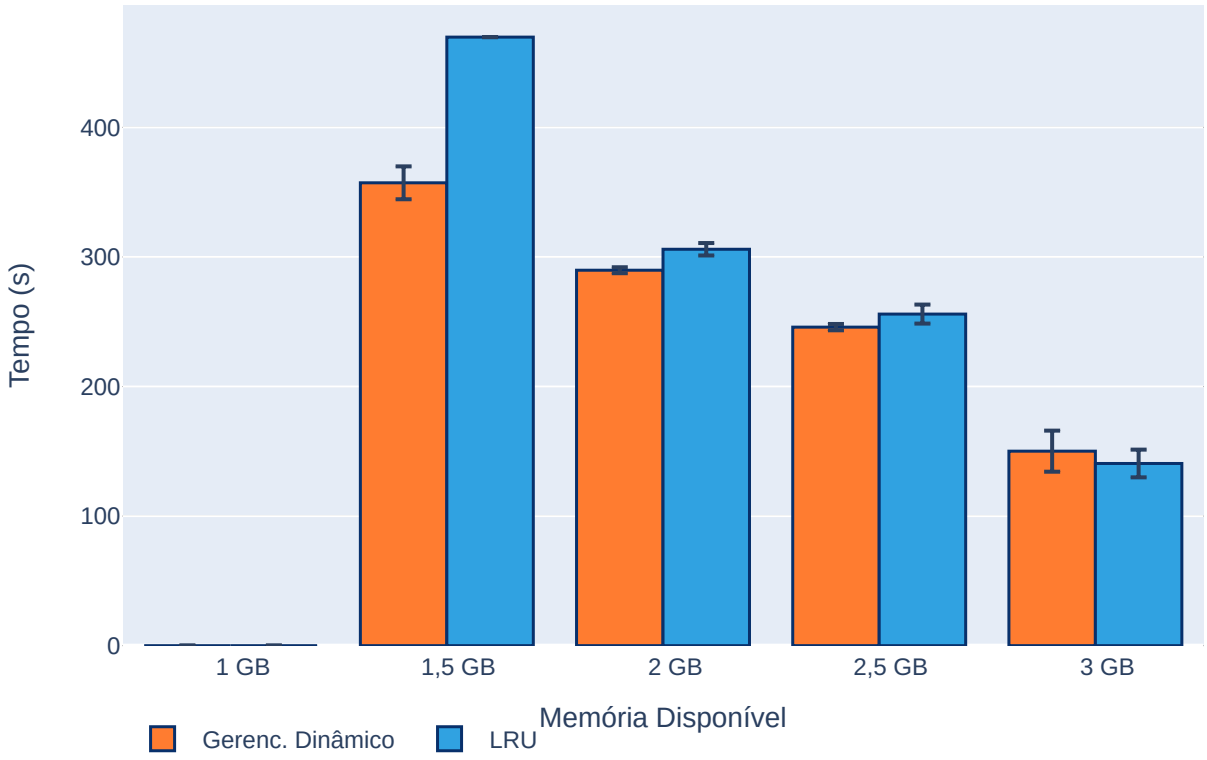
Os resultados obtidos, sendo estes a média aritmética de 20 execução em cada configuração, são demonstrados na Tabela 10 e na Figura 14. A Tabela 11 demonstra a quantidade média de operações de adição e remoção de blocos da memória durante a execução.

Tabela 10 – Tempos de Execuções Obtidos no *Benchmark Logistic Regression*

Algoritmo	Memória Disponível	Tempo de Execução (em s)	Desvio Padrão (em s)
LRU	1 GB	0	0
Gerenc. Dinâmico		0	0
LRU	1,5 GB	469,67*	0,0*
Gerenc. Dinâmico		357,22	12,69
LRU	2 GB	305,94	4,80
Gerenc. Dinâmico		289,76	2,30
LRU	2,5 GB	255,91	7,34
Gerenc. Dinâmico		245,84	2,44
LRU	3 GB	140,60	10,70
Gerenc. Dinâmico		150,11	15,85

Analisando a Tabela 10, na configuração com 1 GB de memória total disponível, tanto o algoritmo LRU quanto o modelo de Gerenciamento Dinâmico não conseguiram terminar a

Figura 14 – Tempos de Execução do *benchmark Logistic Regression*.



Fonte: Próprio autor

execução da aplicação. Nestes casos, uma exceção de *OutOfMemoryError* é lançada pela JVM, indicando uma alta sobrecarga no tempo de execução do *Garbage Collector* (GC) e interrompendo a execução do *benchmark*, sendo este comportamento observado durante as 20 execuções do *benchmark*, em ambos os algoritmos. É importante ressaltar que todas as execuções foram realizadas utilizando o algoritmo padrão de GC provido pela Oracle Java JDK 1.8.

Observando os resultados atingidos, nota-se uma singularidade com relação a configuração com capacidade de 1,5 GB de memória total disponível.Nesta configuração, o algoritmo LRU não conseguiu completar nenhuma execução deste *benchmark*. O tempo demonstrado pela Tabela 10 para a configuração de 1,5 GB do LRU é aquele capturado pelo HiBench para execução de 29 dos 42 *jobs* da aplicação. Assim, o modelo de Gerenciamento Dinâmico foi, em média, 23,94% mais rápido que o tempo registrado para o LRU, mesmo que este algoritmo não tenha terminado o processamento da aplicação por completo.

Além disso, houve uma grande discrepância na estabilidade do Spark durante a execução do *benchmark* nesta configuração de 1,5 GB, onde o algoritmo LRU conseguiu executar apenas 1 vez dentre todas as tentativas realizadas, impossibilitando o cálculo do desvio padrão do

tempo médio de execução. Assim como o ocorrido na configuração com 1 GB disponível, as 19 execuções restantes foram interrompidas pela JVM indicando sobrecarga nas rotinas do *Garbage Collector*.

Quando dispostos 2 GB de memória total disponível, o Gerenciamento Dinâmico foi 11,82% mais rápido, quando comparado ao LRU nesta mesma configuração e considerado a média das execuções. Em situações com 2,5 GB, ambos os métodos de gerenciamento de memória apresentaram desempenho similar, havendo uma redução de apenas 3,93% no tempo de execução do LRU. Na última configuração avaliada, com 3 GB de memória disponível, o modelo de gerenciamento implementado foi 6,37% mais lento que o LRU, acarretando em uma degradação no desempenho. Esta degradação ocorreu devido ao *threshold* adotado, induzindo o aumento na remoção de blocos da memória a fim de evitar o preenchimento completo da memória.

A Tabela 11 tem como objetivo demonstrar o número de inserções e remoções de dados na memória, durante a execução do *benchmark*. Na configuração de 1 GB de memória disponível, nenhuma informação relativa às operações na memória foi capturada, visto que ambos algoritmos não conseguiram finalizar a execução do *benchmark*.

Quando utilizada a configuração com 1,5 GB de memória disponível, não foi possível obter os dados relativos ao comportamento da memória do algoritmo LRU, uma vez que este algoritmo não conseguiu completar a execução do *benchmark*. Nas configurações de 2 GB e 2,5 GB, é possível perceber que o modelo de Gerenciamento Dinâmico realizou menos operações de adição e remoção de partições de RDDs na memória, se comparado ao LRU.

Tabela 11 – Número de Remoções e Inserções de Dados na Memória do *benchmar Logistic Regression*

Algoritmo	Memória	Partições de RDDs Adicionadas	Partições de RDDs Removidas	Variáveis de Broadcast Adicionadas	Variáveis de Broadcast Removidas
LRU	1 GB	-	-	-	-
Gerenc. Dinâmico		-	-	-	-
LRU	1,5 GB	-	-	-	-
Gerenc. Dinâmico		2119	2080	685	628
LRU	2 GB	2317	2270	762	728
Gerenc. Dinâmico		1929	1882	762	710
LRU	2,5 GB	1096	1032	762	734
Gerenc. Dinâmico		922	860	762	730
LRU	3 GB	178	101	762	734
Gerenc. Dinâmico		246	169	762	731

A redução na quantidade de operações realizada na memória demonstra que o algoritmo utilizado pelo modelo de Gerenciamento Dinâmico consegue priorizar os dados de RDDs na memória, mantendo-os por mais tempo em memória quando comparado ao LRU. Essa característica é demonstrada pelo ordem em que os blocos são removidos da memória, onde o algoritmo utilizado no Gerenciamento Dinâmico opta por remover variáveis de *broadcast* antes de iniciar a remoção de dados de RDDs.

No LRU, ao receber uma variável de *broadcast*, o algoritmo armazena esta variável ao final da lista de blocos mantidos na memória. Deste modo, garante-se que deve ser removida por último, uma vez que esta variável teve o acesso mais recente. Se uma nova partição de um RDD precisa ser armazenada, o LRU irá remover uma partição de dados a fim de liberar espaço para a nova partição que deve ser armazenada.

No algoritmo utilizado no Gerenciamento Dinâmico, ao receber um variável de *boradcast*, o algoritmo armazena esta variável no início da lista de blocos mantido em memória. Isto ocorre porque estas variáveis não possuem frequência de reutilização dentro do *job* gerado pelo *DAGScheduler*, tendo o valor 1 como padrão. Assim, em situações onde uma nova partição de um RDD precisa ser armazenada, o modelo dinâmico irá remover primeiro as variáveis de *broadcast* e depois remove as partições de dados. Consequentemente, mais partições de RDDs podem ser mantidas em memória.

Na configuração com 3 GB, o modelo de gerenciamento removeu mais blocos de RDDs que o LRU devido ao *threshold* adotado pelo modelo. Deste modo, a necessidade de manter uma fração da memória sempre livre, imposta pelo Agente de Monitoramento acarretou em remover mais partições que o necessário. Como consequência, o Spark teve que recomputar as partições perdidas, acarretando em uma degradação de 6,37% no desempenho do *framework*.

Assim, ao sumarizar a análise dos dados dispostos nas Tabelas 10 e 11, é possível perceber que ambos algoritmos fizeram uso intensivo da memória disponível, demonstrado pela grande quantidade de operações de adição e remoção de dados da memória. Entretanto, a ordem em que esses blocos são removidos é diferente, onde o algoritmo utilizado no Gerenciamento Dinâmico consegue manter uma maior quantidade de partições em memória. Como consequência, observa-se uma redução no tempo necessário para processar o *benchmark* visto que o número de recomputações é reduzido, em especial em situações onde a memória encontra-se bem restringida, como é o caso onde utilizou-se 1,5 GB, 2 Gb e 2,5 Gb como configuração de memória.

Por fim, em situações onde a memória é suficiente para comportar o *dataset*, a remoção antecipada de blocos da memória pode acarretar em uma degradação no desempenho. Esta degradação pode ser observada na configuração de 3 GB, onde o Gerenciamento Dinâmico removeu e inseriu mais partições de RDDs, aumentando o número de recomputações necessárias.

4 CONSIDERAÇÕES FINAIS

O processamento em memória no Spark é realizado utilizando sua principal abstração: o RDD. Uma vez computado um RDD, este pode ser armazenado em *cache* na memória principal, a fim de evitar sua recomputação a cada acesso. Em situações de sobrecarga da memória, o Spark utiliza o algoritmo LRU para remover partições de RDDs da memória, o qual considera que uma partição frequentemente acessada no passado tende a ser acessada novamente em um futuro próximo. Entretanto, há situações onde algoritmo LRU pode acarretar em uma degradação no desempenho do sistema, como em situações onde há acesso cíclico à memória e a quantidade de dados é maior que o espaço disponível. Este problema pode ser detectado no Spark durante o processamento de aplicações onde há reuso de dados na aplicação.

Visando otimizar o LRU no processamento de aplicações onde há reuso de dados, este trabalho teve por objetivo propor um modelo de Gerenciamento Dinâmico de Memória em Aplicações com Reuso de Dados no Apache Spark. O modelo proposto é composto por dois principais componentes, sendo estes (1) uma política de gerenciamento da memória, responsável por gerir as partições de RDDs mantidas em memória e; (2) um agente externo de monitoramento, a fim de acompanhar a execução da aplicação Spark.

A política de gerenciamento implementada consiste em uma otimização do algoritmo LRU, com o objetivo de agregar a localidade temporal juntamente com a Frequência de Reutilização de cada RDD. Para tanto, o algoritmo analisa o grafo de dependências gerado pelo *job* a fim de identificar a frequência em que cada RDD era reutilizado. Já o agente externo visava monitorar a aplicação Spark, a fim de analisar os dados obtidos e prever a necessidade de remoção de partições da memória. Este agente consiste em uma aplicação Java responsável por acompanhar o andamento da execução da aplicação, bem como a quantidade de memória utilizada para realizar o processamento da aplicação Spark em execução.

A decisão sobre a necessidade de remover partições da memória é realizada utilizando dois critérios: um *threshold* de ocupação da memória e o *status* atual da execução da aplicação. Uma vez identificada a necessidade de realizar a liberação de espaço em memória principal, o agente notifica o nodo cuja memória encontra-se sobrecarregada. Uma vez notificado, o nodo Spark fica responsável por executar a remoção de partições da memória. A notificação do nodo Spark é realizada utilizando o Apache ZooKeeper, onde o Agente de Monitoramento manipula o *znode* associado a cada Spark *Executor* do *cluster*.

O modelo de Gerenciamento Dinâmico foi validado através de experimentos utilizando a plataforma Grid'5000 com os *benchmarks PageRank, K-Means e Logistic Regression*. Além disso, foram adotadas as configurações de 1 GB, 1,5 GB, 2 GB, 2,5 GB e 3 GB de memória disponível no Spark. Os experimentos tinham como objetivo analisar o impacto do método de gerenciamento da memória no tempo de execução final da aplicação, comparando o modelo de Gerenciamento Dinâmico implementado pelo trabalho com o algoritmo padrão do Spark, o LRU.

Os resultados obtidos demonstraram que a solução desenvolvida apresenta resultados satisfatórios, podendo alcançar um desempenho superior ao LRU. Em cenários onde é possível prever a necessidade de memória para novos *jobs*, como no *benchmark Logistic Regression*, o modelo de Gerenciamento Dinâmico pode ser até 23,94% mais rápido que o algoritmo LRU nas mesmas condições.

Uma característica evidenciada é o melhor aproveitamento da memória, quando utilizada a estratégia de remoção de partições de RDDs de maneira antecipada, implementada pelo modelo de gerenciamento descrito por este trabalho. Esta característica é salientada no *benchmark PageRank* onde o modelo de Gerenciamento Dinâmico reduziu em até 34% o tempo de execução deste *benchmark* com 1 GB de memória.

Outra evidência do melhor aproveitamento da memória pode ser encontrada ao utilizar o *benchmark Logistic Regression* no cenário com 1,5 GB de memória disponível. Com esta configuração de memória, o modelo de gerenciamento implementado apresentou uma maior estabilidade na execução, possibilitando a execução do *benchmark* na configuração de 1,5 GB devido ao melhor aproveitamento da memória. Além disso, o uso deste modelo proporcionou a redução do tempo médio de execução. Deste modo, o emprego do modelo de Gerenciamento Dinâmico, o qual visa a utilização de métricas obtidas da aplicação em execução para efetuar o gerenciamento da memória do Spark, pode reduzir o tempo médio de execução de aplicações onde há reutilização de dados.

Este trabalho apresentou um modelo de Gerenciamento Dinâmico da Memória utilizado pelo Apache Spark no processamento de aplicações com Reuso de Dados, sendo estas aplicações um dos grandes focos do *framework*. Entretanto, houve situações onde não foi possível determinar a necessidade de remoção de dados da memória de maneira antecipada, uma vez que o *job* da aplicação era composto por apenas um estágio. Deste modo, sugere-se como trabalho futuros a implementação de um sistema de histórico da execução. Através deste histórico,

torna-se possível mensurar o fluxo necessário para terminar a execução da aplicação, mesmo que esta aplicação seja composta apenas de estágios com um *job*.

Ademais, um outro grande foco do Spark é o processamento de *streaming* de dados, onde o final do fluxo de dados é desconhecido. Assim, há também a possibilidade do investigação, como trabalhos futuros, do uso do LRU em aplicações onde o tamanho do fluxo de dados é desconhecido. Nestes casos, os requerimentos de memória tornam-se diferentes, uma vez que não é possível saber se há reutilização de dados em um futuro próximo.

REFERÊNCIAS

BENESTY, J. et al. Pearson correlation coefficient. In: **Noise reduction in speech processing**. [S.l.]: Springer, 2009. p.1–4.

BOLZE, R. et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. **The International Journal of High Performance Computing Applications**, [S.l.], v.20, n.4, p.481–494, 2006.

CHAMBERS, B.; ZAHARIA, M. **Spark**: the definitive guide: big data processing made simple. [S.l.]: "O'Reilly Media, Inc.", 2018.

COHEN, P.; WEST, S. G.; AIKEN, L. S. **Applied multiple regression/correlation analysis for the behavioral sciences**. [S.l.]: Psychology Press, 2014.

DUAN, M. et al. Selection and replacement algorithms for memory performance improvement in Spark. **Concurrency and Computation: Practice and Experience**, [S.l.], v.28, n.8, p.2473–2486, 2016.

FRAMPTON, M. **Mastering apache spark**. [S.l.]: Packt Publishing Ltd, 2015.

GENG, Y. et al. Lcs: an efficient data eviction strategy for spark. **International Journal of Parallel Programming**, [S.l.], v.45, n.6, p.1285–1297, 2017.

GOLDSCHMIDT, R.; BEZERRA, E.; PASSOS, E. Data mining: conceitos, técnicas, algoritmos, orientações e aplicações. **Rio de Janeiro-RJ: Elsevier**, [S.l.], p.56–60, 2015.

GULATI, S. **Apache Spark 2.x for Java Developers**: explore big data at scale using apache spark 2.x java apis. [S.l.]: Packt Publishing, 2017.

HALOI, S. **Apache ZooKeeper Essentials**. [S.l.]: Packt Publishing Ltd, 2015.

HUANG, S. et al. The HiBench benchmark suite: characterization of the mapreduce-based data analysis. In: DATA ENGINEERING WORKSHOPS (ICDEW), 2010 IEEE 26TH INT. CONFERENCE ON. **Anais...** [S.l.: s.n.], 2010. p.41–51.

HUNT, P. et al. ZooKeeper: wait-free coordination for internet-scale systems. In: USENIX ANNUAL TECHNICAL CONFERENCE. **Anais...** [S.l.: s.n.], 2010. v.8, n.9.

INAGAKI, H. et al. Adaptive Control of Apache Spark's Data Caching Mechanism Based on Workload Characteristics. In: INTERNATIONAL CONFERENCE ON FUTURE INTERNET OF THINGS AND CLOUD WORKSHOPS (FICLOUDW), 2018. **Anais...** [S.l.: s.n.], 2018. p.64–69.

JIANG, S.; ZHANG, X. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. **ACM SIGMETRICS Performance Evaluation Review**, [S.l.], v.30, n.1, p.31–42, 2002.

JUNQUEIRA, F.; REED, B. **ZooKeeper**: distributed process coordination. [S.l.]: "O'Reilly Media, Inc.", 2013.

KARAU, H. et al. **Learning spark**: lightning-fast big data analysis. [S.l.]: "O'Reilly Media, Inc.", 2015.

KARAU, H.; WARREN, R. **High performance Spark**: best practices for scaling and optimizing apache spark. [S.l.]: "O'Reilly Media, Inc.", 2017.

KIM, J. M. et al. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In: SYMPOSIUM ON OPERATING SYSTEM DESIGN & IMPLEMENTATION-VOLUME 4, 4. **Proceedings...** [S.l.: s.n.], 2000. p.9.

LASKOWSKI, J. **Mastering Apache Spark 2 (2017)**. 2017.

LUU, H. **Beginning Apache Spark 2**: with resilient distributed datasets, spark sql, structured streaming and spark machine learning library. [S.l.]: Apress, 2018.

MCAFEE, A. et al. Big data: the management revolution. **Harvard business review**, [S.l.], v.90, n.10, p.60–68, 2012.

OUSSOUS, A. et al. Big Data technologies: a survey. **Journal of King Saud University-Computer and Information Sciences**, [S.l.], v.30, n.4, p.431–448, 2018.

ROBINSON, J. T.; DEVARAKONDA, M. V. **Data cache management using frequency-based replacement**. [S.l.]: ACM, 1990. v.18, n.1.

SPARK, A. **DAGSCheduler**. Disponível em: <https://github.com/apache/spark/blob/branch-2.2/core/src/main/scala/org/apache/spark/scheduler/DAGScheduler.scala>. Acesso em: 20 out. 2019.

SPARK, A. **UnifiedMemoryManager**. Disponível em: <https://github.com/apache/spark/blob/branch-2.2/core/src/main/scala/org/apache/spark/memory/UnifiedMemoryManager.scala>. Acesso em: 25 out. 2019.

SPARK, A. **BlockManager**. Disponível em: <https://github.com/apache/spark/blob/branch-2.2/core/src/main/scala/org/apache/spark/storage/BlockManager.scala>. Acesso em: 02 nov. 2019.

SPARK, A. **MemoryStore**. Disponível em: <https://github.com/apache/spark/blob/branch-2.2/core/src/main/scala/org/apache/spark/storage/memory/MemoryStore.scala>. Acesso em: 20 out. 2019.

WANG, K.; ZHANG, K.; GAO, C. A new scheme for cache optimization based on cluster computing framework spark. In: INTERNATIONAL SYMPOSIUM ON COMPUTATIONAL INTELLIGENCE AND DESIGN (ISCID), 2015. **Anais...** [S.l.: s.n.], 2015. v.1, p.114–117.

XU, E.; SAXENA, M.; CHIU, L. Neutrino: revisiting memory caching for iterative data analytics. In: HOTSTORAGE. **Anais...** [S.l.: s.n.], 2016.

YANG, Z. et al. Intermediate data caching optimization for multi-stage and parallel big data frameworks. In: IEEE 11TH INTERNATIONAL CONFERENCE ON CLOUD COMPUTING (CLOUD), 2018. **Anais...** [S.l.: s.n.], 2018. p.277–284.

ZAHARIA, M. et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: USENIX CONFERENCE ON NETWORKED SYSTEMS DESIGN AND IMPLEMENTATION, 9. **Proceedings...** [S.l.: s.n.], 2012.

ZECEVIC, P.; BONACI, M. **Spark in Action**. [S.l.]: Manning Publications Co., 2016.

ZHANG, M. et al. Intelligent rdd management for high performance in-memory computing in spark. In: INTERNATIONAL CONFERENCE ON WORLD WIDE WEB COMPANION, 26. **Proceedings...** [S.l.: s.n.], 2017. p.873–874.