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

ARQUITETURA DE CONFIGURAÇÃO DINÂMICA PARA A TÉCNICA DE CHECKPOINT EM FRAMEWORKS DE PROCESSAMENTO DISTRIBUÍDO

DISSERTAÇÃO DE MESTRADO

Paulo Vinicius Mendonça Cardoso

Santa Maria, RS, Brasil

2019

ARQUITETURA DE CONFIGURAÇÃO DINÂMICA PARA A TÉCNICA DE CHECKPOINT EM FRAMEWORKS DE PROCESSAMENTO DISTRIBUÍDO

Paulo Vinicius Mendonça Cardoso

Dissertação apresentada ao Curso de Mestrado do Programa de Pós-Graduação em Ciência da Computação (PPGCC), Área de Concentração em Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

Orientadora: Profª. Drª. Patrícia Pitthan de Araújo Barcelos

Santa Maria, RS, Brasil

2019

Universidade Federal de Santa Maria
Centro de Tecnologia
Programa de Pós Graduação em Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

ARQUITETURA DE CONFIGURAÇÃO DINÂMICA PARA A TÉCNICA DE CHECKPOINT EM FRAMEWORKS DE PROCESSAMENTO DISTRIBUÍDO

elaborada por
Paulo Vinicius Mendonça Cardoso

como requisito parcial para obtenção do grau de
Mestre em Ciência da Computação

COMISSÃO EXAMINADORA:

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

Benhur de Oliveira Stein, Dr. (Universidade Federal de Santa Maria)

Luiz Angelo Steffenel, Dr. (Université de Reims Champagne-Ardenne)

Santa Maria, 05 de Agosto de 2019.

“Você vive e aprende. De qualquer forma, você vive. ”
— DOUGLAS ADAMS

RESUMO

Dissertação de Mestrado
Programa de Pós Graduação em Ciência da Computação
Universidade Federal de Santa Maria

ARQUITETURA DE CONFIGURAÇÃO DINÂMICA PARA A TÉCNICA DE CHECKPOINT EM FRAMEWORKS DE PROCESSAMENTO DISTRIBUÍDO

AUTOR: PAULO VINICIUS MENDONÇA CARDOSO
ORIENTADORA: PATRÍCIA PITTHAN DE ARAÚJO BARCELOS
Local da Defesa e Data: Santa Maria, 05 de Agosto de 2019.

O processamento de dados em sistemas computacionais de alto desempenho tornou-se uma tarefa comum dada a grande quantidade de informação gerada atualmente. Conforme a complexidade desses sistemas aumenta, criam-se problemas de confiabilidade e disponibilidade a partir da iminente presença de falhas. Esses fatores motivam a busca por mecanismos de tolerância a falhas para sistemas computacionais. Uma alternativa eficiente é a técnica de *Checkpoint and Recovery* (CR), que busca auxiliar na recuperação pós-falha de um sistema a partir de pontos de verificação previamente salvos. No Apache Hadoop e no Apache Spark – *frameworks* para processamento distribuído de alto desempenho –, o *checkpoint* auxilia em operações de recuperação nos cenários de falha. Porém, a configuração de atributos de *checkpoint* em ambas as ferramentas é estática, de modo que o comportamento da técnica depende de escolhas do desenvolvedor e mudanças em tempo de execução são limitadas. Ou seja, escolhas inapropriadas podem degradar o desempenho e a confiabilidade do sistema. Portanto, este trabalho apresenta uma solução de configuração dinâmica para a técnica de *checkpoint* do Hadoop e do Spark, baseada em um monitoramento de recursos. A proposta é descrita pela arquitetura de configuração dinâmica (*Dynamic Configuration Architecture*, ou DCA), que trabalha partir da definição de métricas de monitoramento. O objetivo da arquitetura é adaptar, em tempo real, os atributos de *checkpoint* de forma eficiente, de acordo com as necessidades dos *frameworks*. Sendo assim, experimentações com e sem falha nos *frameworks* Hadoop e Spark foram executadas a fim de validar a DCA com análises de desempenho e de recuperação. Os resultados mostram que as técnicas de *checkpoint* dinamicamente configuradas pela DCA alcançaram um equilíbrio de desempenho nos cenários de teste. Enquanto execuções sem falhas não geraram uma alta intrusividade, os cenários de falha foram controlados de forma eficiente na maioria dos testes. Além disso, a DCA mostrou uma grande vantagem ao possibilitar o estabelecimento de *checkpoints* em trechos de código indisponíveis ao usuário no Spark. Em trabalhos futuros, otimizações da arquitetura serão desenvolvidos com o refinamento das métricas de monitoramento. Além disso, validações mais completas deverão ser realizadas para um melhor estudo de todos os componentes envolvidos na elaboração da DCA.

Palavras-chave: Tolerância a Falhas. Checkpoint and Recovery. Configuração Dinâmica. Hadoop. Spark. Monitoramento.

ABSTRACT

Master's Dissertation
Undergraduate Program in Computer Science
Federal University of Santa Maria

DYNAMIC CONFIGURATION ARCHITECTURE FOR CHECKPOINT TECHNIQUE ON DISTRIBUTED PROCESSING FRAMEWORKS

AUTHOR: PAULO VINICIUS MENDONÇA CARDOSO
ADVISOR: PATRÍCIA PITTHAN DE ARAÚJO BARCELOS
Defense Place and Date: Santa Maria, August 05th, 2019.

Processing data on High-Performance Computing (HPC) systems is a communal assignment due to large amounts of information being generated. However, reliability and performance problems are created as these systems complexity is increasing. Thus, the search for fault tolerance techniques is important in this context. The Checkpoint and Recovery (CR) fault tolerance technique is widely used for failure recovery based on system stable states that were previously saved. On Apache Hadoop and Apache Spark – distributing and high-performance frameworks –, checkpoint helps on recovery steps after failure events. But checkpoint attribute configuration on both frameworks is static because it depends on the system developer's choices. Also, changes in real-time are not allowed. In this way, inappropriate choices may harm the system's reliability and/or performance. Therefore, this work presents a solution for dynamic configurations for the checkpoint technique on Hadoop and Spark. The purpose is described by the Dynamic Configuration Architecture (DCA) that works with monitoring metrics definitions. The main goal of DCA is to provide real-time adaptations of checkpoint attributes according to the necessity of the framework. Besides the architecture definition, validations were performed on controlled failure scenarios to measure DCA efficiency. Obtained results show that dynamically configured checkpoint techniques reached a balance between performance and reliability (based on recovery time) in most of the tested scenarios. With no failures, executions with DCA did not experience high intrusiveness, as failure scenarios were controlled with fast recovery. Besides, DCA shows a great advantage with the possibility of Spark checkpoint savings even source code parts that are inaccessible from developers. In future works, DCA optimizations will be developed and validated. Monitoring metrics will be improved, as well as the DCA elements. With these optimizations, more accurate validations with several failures and workload scenarios will be performed so the DCA performance can be completely measured.

Keywords: Fault Tolerance, Checkpoint and Recovery, Dynamic Configuration, Hadoop, Spark, Monitoring.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

CF	Cenário de falha
CH	Política estática *checkpoint*
CR	*Checkpoint and Recovery*
DN	$DataNode$
DS	*Dataset* de aprendizado do histórico
FA	Política dinâmica *Failure Awareness*
HDFS	*Hadoop Distributed File System*
HPC	*High Performance Computing*
HRT_{rdd}	Tempo de leitura do *dataset* rdd salvo em *checkpoint*
LL	Política de seleção *Longest Lineage*
LRU	Política de seleção *Least Recently Used*
LT	Política dinâmica *Lineage Threshold*
MAD	Política estática *memory-and-disk*
MO	Política estática *memory-only*
MT	Política dinâmica *Memory Threshold*
MTBF	*Mean Time Between Failures*
NN	$NameNode$
RDD	*Resilient Distributed Dataset*
SNN	$SecondaryNameNode$
TSLF	*Time Since Last Failure*
TE_{rdd}	Tempo de execução do *dataset* rdd
YARN	*Yet Another Resource Navigator*

SUMÁRIO

1 INTRODUÇÃO

A crescente produção de dados digitais gerou a oportunidade da transformação destes dados em conhecimento e informação a partir de procedimentos de processamento computacional. Conforme essa produção cresce, a demanda por sistemas computacionais de alto desempenho (sistemas HPC) exige uma eficiência igualmente maior. Uma grande variedade de arquiteturas de alto desempenho é utilizada para esse propósito. Neste contexto, destaca-se a utilização de sistemas computacionais distribuídos, como *clusters* e *grids* computacionais, que aumentam a eficiência de processamento a partir da distribuição de tarefas em vários nodos.

Para que o potencial de sistemas distribuídos seja altamente desfrutado por desenvolvedores e usuários, diversas ferramentas foram desenvolvidas. Dois dos principais *frameworks* usados para o processamento distribuído de alto desempenho são o Apache Hadoop e o Apache Spark. Nestes *frameworks*, uma aplicação é dividida em tarefas, que são repartidas e enviadas ao nodo de um ambiente distribuído. Ambos os *frameworks* são auxiliados pelo sistema de arquivos distribuído executado no Hadoop: o *Hadoop Distributed File System* (HDFS), que suporta o armazenamento de grandes quantidades de dados (*Big Data*) com o mesmo conceito de distribuição, dividindo arquivos em blocos (WHITE, 2015).

Porém, ainda que estes *frameworks* e outras ferramentas de processamento distribuído contribuam com eficiência de processamento, algumas limitações também se fazem presentes. Fatores como a confiabilidade e a disponibilidade de um sistema são prejudicados à medida em que a arquitetura de um sistema torna-se complexa. Elementos computacionais – quando observados de forma individual – são projetados com um alto nível de disponibilidade. Assim, o tempo médio entre falhas de um único elemento pode ser medido em anos. Porém, quando vários elementos são usados para formar um sistema computacional mais complexo e heterogêneo, a probabilidade de que um destes elementos falhe acaba aumentando. Ou seja, o tempo médio entre falhas do sistema como um todo torna-se menor (EGWUTUOHA et al., 2013).

Uma vez que falhas em sistemas HPC são consideradas inevitáveis (GHIT; EPEMA, 2017), é importante que mecanismos para prevenir, detectar e tratar esse tipo de evento sejam implementados. Sendo assim, a aplicação de técnicas de tolerância a falhas é essencial para evitar erros computacionais decorrentes de falhas. Uma técnica tolerância a falhas bastante utilizada é a recuperação de erros, a qual pode ocorrer por avanço ou por retorno (LAPRIE, 1985).

No contexto da recuperação de erros por retorno, uma eficiente técnica é utilizada por diversos sistemas, essencialmente os sistemas de processamento distribuído. O *Checkpoint and Recovery* (CR, ou apenas *checkpoint*) corresponde a uma técnica de recuperação de erros por retorno cujo objetivo é conduzir o sistema a um estado estável anterior ao evento de falha. O CR consiste em duas fases: o estabelecimento preventivo de *checkpoints* e a recuperação, que acontece após a falha e consiste em recuperar o andamento normal do sistema a partir do *checkpoint* mais recentemente estabelecido.

Uma implementação da técnica de *checkpoint* pode ser encontrada no Apache Hadoop, cuja finalidade é auxiliar o procedimento de recuperação do elemento mestre do HDFS: o $NameNode$. Uma vez que o $NameNode$ funciona como um elemento central de controle do HDFS, eventos de falha nesse elemento prejudicam diretamente a confiabilidade do *framework*, além de comprometerem a disponibilidade dos dados armazenados. O *checkpoint* atua no sentido de salvar o contexto estável do $NameNode$ periodicamente, antes de um evento de falha. Assim, quando o $NameNode$ falha, sua recuperação leva em conta o último *checkpoint* salvo para retomar sua execução.

O Apache Spark também possui uma implementação do *checkpoint* como técnica de tolerância a falhas. Nesse *framework*, os dados de processamento são baseados em um conceito de *dataset* que pode ser armazenado na memória principal, a fim de garantir leituras e escritas mais rápidas. Os dados são representados por RDDs: *Resilient Distributed Datasets*. Cada partição de um RDD mantém informações acerca de sua criação, através da noção de *lineage* (linha do tempo). Assim, partições perdidas podem ser recuperadas pelas informações contidas em outras partições. Nesse cenário, *checkpoints* são usados para o armazenamento de RDDs em repositórios seguros (como o HDFS), de modo a melhorar o desempenho de recuperações pós-falha. Nos casos em que o espaço em memória é limitado e, essencialmente, quando o custo de recomputação é grande, o *checkpoint* contribui com recuperações mais rápidas.

1.1 Justificativa

O uso de *checkpoints* no Hadoop e no Spark mostra-se importante em casos de falha ao aumentar a garantia de confiabilidade e disponibilidade de dados por meio de recuperações eficientes. Porém, a configuração de atributos e comportamentos do *checkpoint* nesses *frameworks* apresenta limitações que podem comprometer o propósito da técnica. Os *checkpoints* estáticos – configurados de forma estática e/ou manual – podem ser prejudiciais para o desempenho e

para a confiabilidade de ambos os *frameworks*. Ou seja, se configurações do CR forem definidas visando-se apenas a confiabilidade, os níveis de intrusividade podem ser altos, dado que um maior número de salvamentos de *checkpoint* deve ser estabelecido. Por outro lado, a preocupação com o desempenho pode tornar os arquivos de *checkpoints* defasados, de modo que a etapa de recuperação deverá ser sobrecarregada.

Apesar de suas propostas eficientes no contexto da tolerância a falhas, tanto o Hadoop como o Spark possuem sérias limitações em suas implementações de *checkpoint* devido a configurações estáticas. No Hadoop, *checkpoints* com períodos estaticamente configurados interferem no desempenho do sistema. Como mudanças no período entre *checkpoints* requerem o reinício do *framework*, períodos ideais são difíceis de serem definidos. Já no Spark, além da característica estática, o *checkpoint* é manual. Isto é, os desenvolvedores devem especificar manualmente, diretamente via código, quando e quais *datasets* devem ser marcados para *checkpoint*.

Visto que diferentes aplicações possuem demandas exclusivas, o procedimento de *checkpoint* pode se comportar de maneira distinta em cada situação. Além disso, o sistema pode apresentar variações de comportamento devido a fatores como falhas, cargas de dados e a concorrência fomentada por mais de um cliente consumindo os mesmos serviços. Por isso, há uma necessidade de alterar o comportamento da técnica de *checkpoint*. A escolha de configurações ideais – que ofereçam alta confiabilidade mas não interfiram no desempenho das aplicações – consiste em um grande desafio. Diante de características estáticas, e devido à inflexibilidade de mudanças no comportamento do *checkpoint* dos *frameworks* citados, escolhas inapropriadas levam à perda de eficiência da técnica. Nesse caso, o propósito da aplicação de uma técnica de tolerância a falhas é perdido.

1.2 Objetivos

Este trabalho propõe uma abordagem para resolver os problemas de *checkpoints* estáticos através da definição de uma arquitetura de configuração dinâmica (*Dynamic Configuration Architecture*, ou DCA) para a definição de atributos de *checkpoint*. O objetivo essencial da DCA é tornar a utilização do *checkpoint* mais eficiente, de modo que a tolerância a falhas proposta pela técnica de CR seja melhor explorada enquanto sua intrusividade é controlada. Para isso, a arquitetura proposta oferece adaptações em tempo real para implementações da técnica de CR no Hadoop, além de definir *checkpoints* em *datasets* de forma automática no Spark.

As adaptações para ambas as ferramentas levam em consideração um monitoramento de recursos centralizado, com a definição de um módulo para monitoramento (*monitor*) e outro para a coordenação de atributos (*coordinator*). O monitoramento é capaz de verificar a situação dos elementos do sistema como um todo, através de agentes de monitoramento externos (*external agents*). Além disso, monitores internos (*internal agents*) fazem a coleta dos níveis de utilização e desempenho dos *frameworks* e também fazem parte da proposta. Atuando como o elemento central do *monitor*, o *supervisor* recebe *reports* de *agents* e promove mudanças em configurações quando necessário, de acordo com métricas estabelecidas.

Todos as mudanças de contexto identificadas pelo *supervisor* passam pelo elemento *coordinator*, implementado com o auxílio da ferramenta Apache Zookeeper. Neste elemento, atributos de configuração já usados são armazenados para um controle interno da DCA. O Zookeeper também oferece o serviço de *watchers*, fundamental para uma comunicação bi-lateral entre o *coordinator* e o *monitor*. Como forma de facilitar a estimativa de custos e fatores de utilização, os quais são difíceis de obter de forma antecipada, o *coordinator* também propõe um esquema de histórico. Neste esquema, o comportamento do sistema pode ser estimado de acordo com estatísticas previamente coletadas, sob a forma de um modelo de aprendizagem.

A partir da definição e implementação da arquitetura nos *frameworks* Hadoop e Spark, este trabalho realiza validações com foco em desempenho e confiabilidade. Como forma de testar a eficiência do mecanismo dinâmico, diferentes *benchmarks* são executados com diversas configurações estáticas (isto é, configurações padrão dos *frameworks*) e configurações dinâmicas com variações nas políticas e métricas de monitoramento usadas. Cenários de falha são descritos a fim de testar a atuação das configurações quando eventos de falha e procedimentos de recuperação estão presentes.

1.3 Estrutura do texto

O trabalho está dividido em cinco capítulos. No Capítulo 2, é apresentada a fundamentação teórica a respeito da técnica de *checkpoint* e os *frameworks* usados: Apache Hadoop e Apache Spark. O Capítulo 3 descreve todos os detalhes de projeto e implementação da DCA. No Capítulo 4, as validações da arquitetura são exploradas. Além disso, os resultados obtidos em execuções são discutidos e avaliados. Por fim, o Capítulo 5 apresenta as considerações finais do trabalho, apontando-se conclusões e perspectivas de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Dentre as técnicas de tolerância a falhas, destaca-se o uso do *Checkpoint and Recovery* (CR) em sistemas computacionais de alto desempenho. Este capítulo faz um estudo a respeito do *checkpoint* e de suas implementações nos *frameworks* Hadoop e Spark. Na seção 2.1, é explorada a técnica de *checkpoint*, bem como suas principais vantagens e funcionalidades. A seção 2.2 fala sobre o Hadoop e como o seu sistema de arquivos distribuído usa a técnica de *checkpoint*. Finalmente, a seção 2.3 fala sobre o funcionamento do Apache Spark, com ênfase na técnica de *checkpoint* em sua abstração de dados em memória.

2.1 Checkpoint

Checkpoint and Recovery (CR) é uma técnica de tolerância a falhas reativa, classificada como técnica de recuperação por retorno (*backward error recovery*). O CR tem como ideia principal a recuperação do estado falho de um sistema através de um contexto estável previamente salvo (EGWUTUOHA et al., 2013). Essa recuperação visa prevenir erros computacionais consequentes das falhas. Os contextos estáveis são salvos periodicamente e armazenados de forma segura para uma posterior recuperação do sistema.

O procedimento usado pelo CR segue duas etapas essenciais: a criação de *checkpoints* e a recuperação em momentos pós-falha. De forma geral, quando o sistema está executando sem a presença de falhas, o CR é responsável por salvar as informações atuais a respeito do estado do sistema em arquivos de *checkpoint*. Em cenários que o *checkpoint* é feito de forma periódica, o intervalo escolhido para esta operação é importante, de modo que períodos mais espaçados podem comprometer a confiabilidade da aplicação e intervalos menores geram sobrecargas.

Os casos em que a técnica de *checkpoint* se mostra vantajosa estão relacionados a aplicações de longa duração (*long-running applications*) e com processamento intensivo de dados. Nesses casos, altos níveis de tempo de execução, complexidade e quantidade de operações em conjuntos de dados são intensamente observados. Desta forma, o estabelecimento de *checkpoints* surge como uma alternativa para procedimentos de recuperação mediante falhas.

Uma das vantagens do CR, em relação a técnicas como redundância (BARTLETT; SPAINHOWER, 2004) e migração (LIU et al., 2013) é o custo envolvido nas operações. O custo energético é potencialmente menor que outras técnicas como a redundância física, uma

vez que o *checkpoint* não requer *hardwares* adicionais (ROMAN, 2002). Além disso, o custo de implementação não é tão alto quanto em técnicas de migração que necessitam de um eficiente previsor de falhas, por exemplo. Ao contrário, implementações de CR necessitam pouco recurso adicional e podem ser desenvolvidas de forma simples.

2.1.1 Limitações

Todavia, existem muitos desafios relacionados à técnica de *Checkpoint and Recovery*. Em falhas de implementação da aplicação, por exemplo, um trecho de código deve gerar um evento de falha e o processo de *recovery* deve iniciar. Mas ao executar o código falho novamente, uma nova falha irá surgir, repetindo esse processo até que uma intervenção externa seja realizada. A intervenção externa para a resolução de problemas é encarada como um grande problema, sobretudo quando o funcionamento da técnica depende exclusivamente de usuários ou desenvolvedores. Este é o caso do Apache Spark, cuja implementação de CR é dependente da intervenção do desenvolvedor para que a etapa de salvamento de *checkpoints* aconteça.

Além disso, métodos tradicionais (*disk-based checkpoint*) de CR utilizam o disco como forma principal de armazenamento dos *checkpoints*, devido a grande quantidade de espaço disponível e o pequeno *overhead* de espaço consumido. Contudo, a leitura e escrita em disco se tornam mais frequentes à medida que aumentamos o nível de confiabilidade desejado, comprometendo-se a eficiência do sistema em relação ao seu desempenho computacional. Por outro lado, a escolha por um período mais longo de salvamento de *checkpoints* gera uma quantidade maior de informações a serem lidas e armazenadas, ainda que em uma frequência menor.

Para evitar a degradação de desempenho causada pelos salvamentos de *checkpoint* baseados em disco, novas soluções tem sido buscadas, como é o caso do *diskless checkpoint*. Esse tipo de solução usa meios de armazenamento alternativos e procedimentos para redução da quantidade de dados em *checkpoints*, a fim de criar adaptações para um novo contexto que use o disco de forma menos frequente. Uma forma de armazenamento amplamente utilizada é a memória principal, dado o seu eficiente tempo de acesso aos dados. A busca por técnicas de *checkpoint* em memória é vista como uma alternativa promissora e tem sido encarada como um diferencial para o desempenho de muitas aplicações.

Porém, apesar do ganho em desempenho, novos desafios são incluídos ao utilizar-se a memória. Ao contrário do disco, o espaço em memória é consideravelmente limitado em muitos casos, sobretudo em sistemas de baixo custo. Aplicações que usam a memória de forma

abundante podem sofrer com a falta de espaço proveniente do salvamento de *checkpoints*. Além disso, para garantir confiabilidade em um ambiente volátil, *checkpoints* em memória devem buscar soluções como a replicação de dados. O Spark surge como uma alternativa interessante no contexto de armazenamento eficiente de dados e *checkpoints*, através de *datasets* que podem ser armazenados em memória sem a necessidade de replicação.

2.2 Apache Hadoop

O Apache Hadoop é um projeto *open source* voltado para o processamento distribuído de grandes quantidades de dados (WHITE, 2015). O Hadoop oferece um eficiente sistema de distribuição de dados e aplicações em arquiteturas de alto desempenho, como *clusters* e *grids*. O conceito essencial do Hadoop é mover a aplicação até os dados – evitando mover os dados em si – para obter eficiência de processamento (JAIN; GOYAL, 2017).

A arquitetura do Hadoop, em sua versão utilizada por este trabalho (v2.7.3), é composta por diversos módulos, dentre os quais destacam-se: o sistema de arquivos distribuído (HDFS) e o *framework* para manutenção do ambiente e da aplicação (YARN).

2.2.1 YARN

Para o gerenciamento dos recursos disponíveis e para a alocação de tarefas, o Hadoop utiliza a ferramenta *Yet Another Resource Navigator* (YARN). A ideia do YARN é dividir as duas funcionalidades oferecidas em um $ResourceManager$ (RM) global, para tratar dos recursos do ambiente, e um $ApplicationMaster$ (AM) para cada aplicação específica (FOUNDATION, 2019b). Desta forma, o Hadoop oferece uma plataforma de execução mais eficiente e flexível, permitindo a execução de aplicações com diversos modelos de programação.

O $ResourceManager$ atua como um gerenciador centralizado, comunicando-se com $NodeManagers$ executados em cada nó do ambiente e realizando a função de escalonamento de recursos às aplicações. Um $NodeManager$ reporta a situação do seu nó ao RM através do gerenciamento de *containers*. Já o $ApplicationMaster$ é responsável por negociar recursos requisitados pelas aplicações ao RM, além de trabalhar com os $NodeManagers$ e seus *containers* para a execução de tarefas.

Dentre os modelos de computação suportados pelo YARN, destaca-se o paradigma *MapReduce*. Esse modelo é voltado para a construção de aplicações distribuídas que operam com

grandes quantidades de dados (DEAN; GHEMAWAT, 2010), sendo inicialmente uma parte essencial do Hadoop. Uma aplicação *MapReduce* possui, além de outros métodos opcionais, as funções de mapeamento (*map*) e redução (*reduce*) baseadas em linguagens funcionais.

Na etapa de mapeamento, a aplicação é preparada com uma divisão de tarefas através do ambiente utilizado, dividindo-se a entrada e definindo-se o trabalho de cada nó. A divisão da entrada se dá pela quantidade de arquivos e/ou pelo do tamanho dos dados. Também pode-se definir o tamanho que cada divisão pode assumir. A saída das funções *map* são reunidas na etapa de redução, onde um novo processamento é executado caso necessário, e a saída final da aplicação é gerada.

2.2.2 HDFS

Os dados no Apache Hadoop são armazenados em um sistema de arquivos distribuído denominado *Hadoop Distributed File System* (HDFS), que foi projetado para oferecer suporte a arquivos de grandes dimensões. Um arquivo no HDFS é dividido em blocos de tamanhos pré-definidos e distribuídos através do ambiente. A distribuição faz com que blocos de um mesmo arquivo possam estar armazenados em diferentes nós, para que o propósito de mover a computação até os nós, implementado pelo Hadoop, seja facilitado.

No HDFS, dados e metadados são armazenados de forma separada. Os metadados são mantidos por um servidor dedicado, chamado $NameNode$ (NN), enquanto um $DataNode$ (DN) é responsável por realizar o armazenamento dos dados. A arquitetura do HDFS é baseada no modelo *1-master N-workers*, frequentemente encontrado em sistemas distribuídos (TANENBAUM; VAN STEEN, 2007). Nesse caso, o *master* é representado pelo $NameNode$ e seus N *workers* são os *DataNodes*, como mostra a Figura 2.1. O $NameNode$ também atende às requisições de aplicações e gerencia os arquivos e $DataNodes$ do HDFS.

A representação de arquivos e diretórios no $NameNode$ é feita por objetos do tipo `INode`, que são armazenados em sua memória local, tornando disponível um mapeamento de arquivos/blocos no HDFS. Para completar o *namespace*, há um outro mapeamento que possui informações sobre a localização dos blocos de cada DN. Em um intervalo de tempo pré-definido (por padrão 4 segundos), os $DataNodes$ realizam uma varredura em seus blocos e reportam o estado para o $NameNode$. Assim, o nó mestre do HDFS possui uma visão sobre a localização de cada bloco do HDFS.

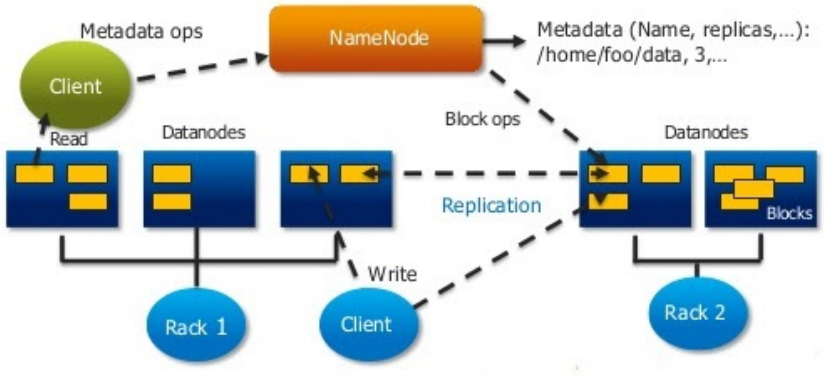


Figura 2.1 – Arquitetura utilizada pelo HDFS (EDUREKA, 2019)

2.2.3 Tolerância a falhas

Para garantir confiabilidade e disponibilidade, o HDFS possui implementações de técnicas de tolerância a falhas em seus processos. As técnicas devem garantir que os serviços oferecidos pelo *framework* sejam mantidos, mesmo quando um evento de falha acontece. Dentre as técnicas de tolerância a falhas implementadas pelo Hadoop, tem-se a replicação de blocos, o envio de mensagens *heartbeat* e o estabelecimento de *checkpoints*.

2.2.3.1 Replicação de Blocos

O HDFS trabalha com a técnica de replicação para assegurar a disponibilidade de todos os seus blocos. O número de réplicas, controlado pelo fator de replicação, define a quantidade de cópias que um mesmo bloco possui no sistema. Dessa forma, dados de $DataNodes$ falhos podem ser recuperados a partir de réplicas armazenadas em outros nós, evitando que as aplicações deixem de funcionar corretamente.

O fator de replicação e o tamanho de um bloco são previamente definidos (respectivamente, 3 réplicas e 128MB), mas podem ser configurados para cada arquivo individualmente. A técnica de replicação protege os dados armazenados no HDFS para até $n-1$ falhas simultâneas, quando define-se um fator de replicação n. O processo de replicação é transparente ao usuário do HDFS, que interage apenas com o primeiro $DataNode$. As réplicas são repassadas pelos $DataNodes$ até que o fator de replicação seja satisfeito.

A localização das réplicas tem um papel importante para a confiabilidade e para o desempenho da replicação. Por padrão, as duas primeiras réplicas são armazenadas em nós que

estejam próximos entre si, de acordo com o conceito de *rack-awareness* (SHAFER; RIXNER; COX, 2010). Assim, o HDFS visa um acesso mais rápido em caso de recuperação. A réplica seguinte é armazenada em um *rack* diferente, com maior foco em confiabilidade.

2.2.3.2 Heartbeat

O envio de mensagens *heartbeat* consiste em uma técnica de tolerância a falhas usada para a detecção de falhas em $DataNodes$ (WHITE, 2015). Periodicamente, os DNs ativos enviam avisos ao NN a fim de atualizá-lo sobre seu estado. Em um aviso, o DN reporta o estado de seus blocos e sinaliza sua plena atividade, com o intuito de mostrar que está ativo. Quando um DN deixa de funcionar, por falhas ou problemas de comunicação, o NN o define como um elemento falho e deixa de enviar instruções ao mesmo.

O intervalo padrão para o envio de mensagens *heartbeat* é de 4 segundos. A detecção de um $DataNode$ falho acontece quando um *timeout* de 10 minutos é atingido sem que o $DataNode$ envie um *heartbeat*. Um DN detectado como falho é considerado fora de serviço e seus blocos são marcados como indisponíveis. Nesse caso, o $NameNode$ deve realizar um procedimento de recuperação do fator de replicação a partir da criação, em outros $DataNodes$ ativos, das réplicas perdidas.

Além da característica tolerante a falhas, o *heartbeat* também serve para reportar informações sobre o armazenamento usado pelo $DataNode$. Informações como o espaço usado em disco e metadados de blocos também são enviados ao $NameNode$. Assim, o *heartbeat* auxilia no controle do $NameNode$ sobre a localização e o estado dos blocos alocados através do HDFS.

2.2.3.3 Checkpoint

O *checkpoint* no Hadoop é implementado para tolerar falhas no HDFS. Apesar de não ser a principal técnica de tolerância a falhas do Hadoop, o salvamento de *checkpoints* é essencial para a manutenção do $NameNode$ e de seus metadados. Assim, o *namespace* do HDFS é replicado em um arquivo chamado `FSImage`, armazenado em disco no sistema de arquivos local do NN. Nesse arquivo, são mantidas informações sobre o mapeamento de blocos e propriedades do sistema.

Para evitar que um novo `FSImage` seja criado a cada operação feita, um *log* de edições (`EditLog`) também é mantido em disco local. Nesse arquivo de *log*, são armazenadas as úl-

timas transações feitas pelo $NameNode$ desde que o último *checkpoint* foi realizado. Desta forma, pode-se notar que o Hadoop implementa uma técnica de *checkpoint* incremental (NAKSINEHABOON et al., 2008).

O estabelecimento de um *checkpoint* consiste na etapa de agrupamento entre o $FSImage$ mais atualizado com o $EditLog$ atual. Para que o $NameNode$ não seja interrompido durante o agrupamento, o Hadoop implementa um elemento cuja função é exclusivamente estabelecer *checkpoints*: o $Secondary\ NameNode$ (SNN). A Figura 2.2 mostra a relação entre NN e SNN. Ainda que possa ser relacionado ao $NameNode$, devido a sua nomenclatura, o SNN não emprega nenhuma função do elemento mestre do HDFS. Ou seja, em caso de falha no NN, o SNN não se torna uma alternativa de recuperação.

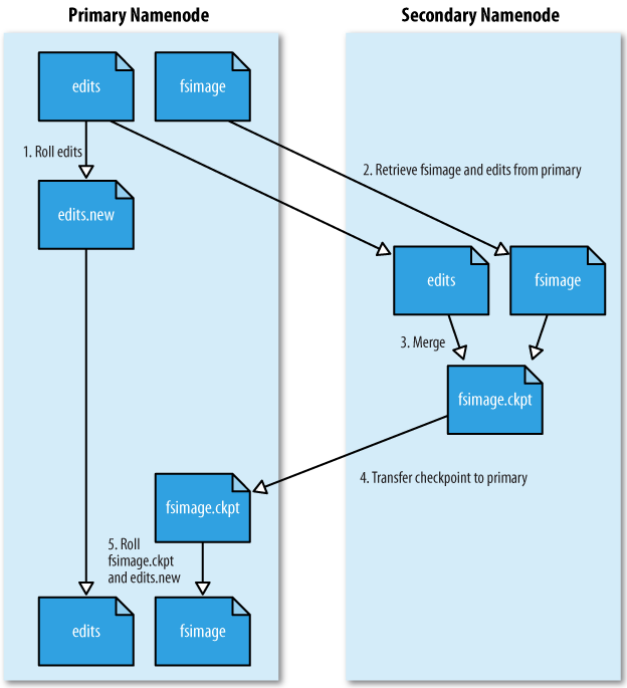


Figura 2.2 – O procedimento de *checkpoint* no Hadoop (WHITE, 2015)

O SNN mantém uma cópia do `FSImage` atual armazenada em seu sistema de arquivos local, de modo que a imagem sempre está de acordo com a imagem armazenada pelo $NameNode$. Em toda requisição por um *checkpoint*, há uma transferência de dados entre os dois elementos, onde o NN envia o arquivo de edições atual ao SNN. Assim que o *checkpoint* é salvo, mais uma transferência se dá pelo envio da nova imagem ao $NameNode$.

Em versões *High Availability* (HA), o Hadoop implementa o $Standby\ NameNode$, que se mantém sincronizado com o $NameNode$ ativo e o substitui em caso de falha. Porém, a sincronização em tempo real é uma tarefa que adiciona sobrecargas ao *framework*, uma vez que

o elemento replicado consome tantos recursos quanto o elemento ativo (LEVITIN et al., 2017).

O salvamento do $FSImage$ em *checkpoint* é feito de forma periódica, mas também pode ser disparado se o HDFS atingir um limite específico de transações. Os atributos *default* são 3600 segundos para o intervalo entre *checkpoints*, e 10 mil operações para o limite de transações. Esses valores podem ser modificados a partir do arquivo de configuração do HDFS ($hdfs - site.xml$). Porém, mesmo que possam ser modificados, os atributos de *checkpoint* do Hadoop são estáticos. Isto é, as modificações não são possíveis em tempo de execução. Sendo assim, uma alteração em um dos atributos requer que todos os serviços do Hadoop, além das aplicações em andamento, sejam interrompidos.

Visto que diferentes aplicações possuem demandas exclusivas, a técnica de *checkpoint* pode se comportar de maneira distinta em cada situação. Por isso, a escolha do intervalo ideal entre *checkpoints* – que ofereça uma alta confiabilidade mas não interfira no desempenho das aplicações – consiste em um grande desafio. *Checkpoints* frequentes tornam recuperações mais rápidas mas adicionam uma alta sobrecarga, ao passo que frequências menores podem comprometer a confiabilidade da técnica (DALY, 2003).

Além disso, intervalos estáticos não oferecem suporte às mudanças de comportamento do ambiente, pois são definidos de forma prévia e não mudam no decorrer da execução. Por isso, a característica estática de configuração dos atributos de *checkpoint* do Hadoop pode ser prejudicial ao próprio *framework*, tanto em relação a sua confiabilidade quanto ao seu desempenho.

2.2.4 Hadoop Metrics

Dentro de seu ambiente de produção, o Hadoop contém uma série de elementos interagindo entre si. Nesse sentido, é importante que a ferramenta e os elementos gerenciadores tenham um acesso facilitado a informações sobre o estado e o desempenho dos outros elementos. No HDFS, por exemplo, o *NameNode* deve certificar-se das informações de $DataNodes$ sob sua responsabilidade para enviar ou conter novas requisições para determinados nodos.

Como forma de facilitar o monitoramento de recursos externamente, o Hadoop oferece uma estrutura chamada *Hadoop Metrics*. Essa estrutura pode ser usada por sistemas de monitoramento para avaliar o estado de execução do Hadoop ou para realizar tarefas de depuração (WHITE, 2015).

O *Hadoop Metrics* foi projetado com três elementos principais. O elemento $MetricsSource$

consiste na definição de uma métrica e da informação a ser coletada. Em sua inicialização, o Hadoop cria um $Source$ para seus principais elementos. Já o $MetricsSystem$ consiste no elemento central do projeto de métricas. Isto é, o $System$ atua como uma interface entre os dados coletados pelo $Source$ e os consumidores definidos pelo terceiro elemento da estrutura – os $MetricsSinks$. Portanto, a recuperação de uma informação por um consumidor externo ($Sink$) consiste no acesso ao $System$ que, por sua vez, acessa as métricas via $Source$.

A requisição de uma ou várias informações disponibilizadas pelo *Hadoop Metrics* é feita através de requisições HTTP do tipo GET. A requisição é retornada com um objeto JSON contendo os dados solicitados organizados por elemento. Por outro lado, pode-se direcionar métricas de forma automática para arquivos. Para isso, deve-se especificar cada direcionamento nos arquivos de configuração do Hadoop.

2.3 Apache Spark

O *framework* Apache Spark é um projeto *open source* que oferece uma plataforma voltada para a computação distribuída, designado para ser rápido e de propósito geral (KARAU; WARREN, 2017). O Spark não é uma solução de armazenamento de dados, mas define-se como um motor computacional (*computational engine*) responsável por realizar tarefas de escalonamento, distribuição e monitoramento de aplicações em máquinas trabalhadoras de um ambiente distribuído (KARAU et al., 2015).

Originalmente, o Spark foi considerado uma extensão de *frameworks*, como o Hadoop, para o suporte a uma maior variedade de modelos e paradigmas de computação. Para suprir limitações de desempenho vistas em tarefas que exigem um reuso constante de dados, principalmente tratando-se de algoritmos iterativos, o Spark oferece um modelo computacional de alto desempenho com características de tolerância a falhas mesmo com o armazenamento de dados em memória.

O uso de processamento em memória é uma das grandes vantagens do Spark, já que tarefas podem explorar esse recurso para garantir rapidez em operações de escrita e leitura de dados (SHI et al., 2015) (VERMA; PATEL, 2016). Apesar disso, o uso de disco no Spark é um importante fator de desempenho para aplicações, uma vez que grande parte das arquiteturas de sistemas computacionais possuem limitações na quantidade de memória disponível. Aplicações com uma quantidade grande de dados para processar tendem a gerar limitações quando todo espaço disponível em memória é usado. Nesse caso, o disco torna-se um elemento chave para a

utilização em aplicações de alto desempenho, sobretudo a partir da escolha eficiente de políticas para gerenciamento dos dados.

2.3.1 Arquitetura

A arquitetura do Spark segue um modelo comum em sistemas distribuídos baseados em *cluster* (TANENBAUM; VAN STEEN, 2007), com a definição de um único mestre (*master*, ou $ClusterManager$) que deve gerenciar um ou diversos trabalhadores (*workers*). O elemento mestre é responsável por atender a requisições de aplicações e, então, alocar recursos para a execução de tarefas (*tasks*) nos trabalhadores. A fim de atender a requisições por recursos, um *worker* cria *executors* que possuem a finalidade de processar as operações submetidas por suas aplicações. A Figura 2.3 exibe a arquitetura do Spark e as comunicações entre os elementos.

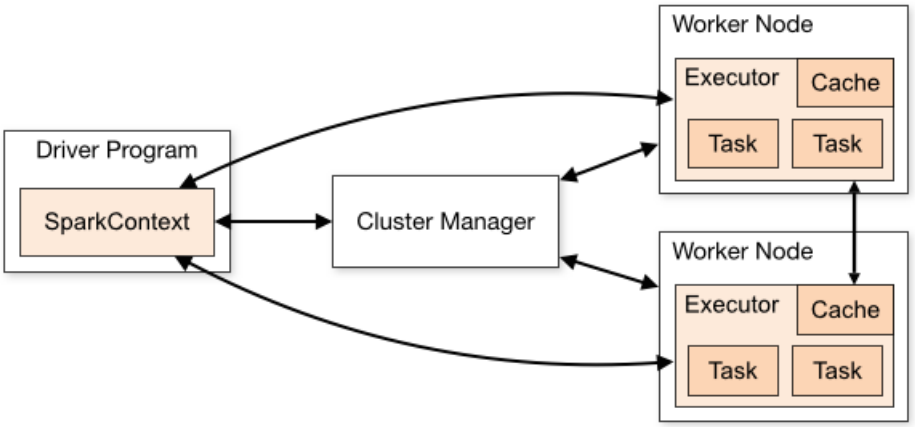


Figura 2.3 – Arquitetura implementada pelo Spark (FOUNDATION, 2019a).

O *master* pode ser implementado por três diferentes gerenciadores de *cluster* (FOUNDATION, 2019a): (a) $StandaloneCluster$, que oferece uma interface de gerenciamento simples e de fácil utilização; (b) Apache Mesos, um gerenciador de propósito geral; e (c) Apache YARN, o gerenciador de recursos na versão 2.x do Hadoop. A escolha do $ClusterManager$ é importante para o cenário de execução do Spark. Nesse sentido, a abordagem $Standalone$ é geralmente usada em cenários menos complexos, enquanto que o YARN e o MESOS são considerados escolhas interessantes em ambientes onde os recursos são compartilhados com outras ferramentas.

Uma vez que o Spark pode trabalhar com mais de um tipo de $ClusterManager$, deve haver um meio de abstração para o escalonamento de recursos não seja afetado pela diferença entre os gerenciadores. Para isso, usa-se uma interface chamada $SchedulerBackend$ (*bac-*

kend), que suporta os três gerenciadores e abstrai suas diferenças. Isto é, toda a comunicação entre as aplicações e o *master* deve passar pelo *backend*, para que o protocolo usado seja o mesmo.

Uma aplicação do Spark possui a definição de um $DriverProgram$ (*driver*), que é responsável por controlar o fluxo de tarefas da aplicação e por iniciá-las nos *executors* alocados pelo *master* (KARAU et al., 2015). O *driver* é executado na mesma máquina em que a aplicação foi submetida e pode ser considerado o ponto de partida da execução, uma vez que carrega o método principal da aplicação. Outro papel importante do *driver* é a negociação de recursos com o *master* a fim de obter *executors* para suas execuções.

O principal componente do *driver* é o Spark Context (*context*), criado para que a aplicação tenha acesso às funcionalidades e serviços do Spark. Além de representar a conexão do *driver* com o *cluster*, o*context* oferece a possibilidade de execução de diversas operações para a criação e o gerenciamento de dados pelo Spark. Uma aplicação tem sua vida útil definida pelo início e pelo fim do objeto $SparkContext$.

Para gerenciar dados em memória, o Spark se aproveita de uma abstração de dados chamada *Resilient Distributed Dataset* (RDD). RDDs são *datasets* que podem ser divididos em partições e distribuídos através do *cluster*. Além da definição dos dados, um RDD armazena a sua relação de dependência com outros *datasets*, formando-se uma linha do tempo (*lineage*) capaz de garantir recuperações em casos de falha de alguma partição a partir da sua reconstrução.

2.3.2 RDD: Resilient Distributed Dataset

O conceito básico do armazenamento de dados no Spark é definido por uma abstração chamada *Resilient Distributed Dataset* (RDD). Essa abstração é definida como uma coleção de dados com característica *read-only*, que é particionada através do ambiente distribuído do Spark (ZAHARIA et al., 2012). Isto é, os RDDs são representações de *datasets* na programação do Spark. Suas elaborações e modificações são realizadas a partir das operações da ferramenta. Em geral, um RDD é dividido em diversas partições, que são armazenadas e processadas por diferentes nós de um *cluster* através dos *executors*.

Os RDDs são objetos cujas estruturas constituídas por 5 componentes principais: uma lista de RDDs precedentes – ou dependências –, um *array* contendo as partições criadas para o *dataset*, a função usada para gerar o RDD, um particionador (opcional) e uma lista de localizações preferidas para o armazenamento de partições. O objeto correspondente a um RDD é

mantido pelo *context* e identificado por um nome e por um identificador único (*id*). Um RDD só deve estar disponível enquanto o*context* de sua aplicação também estiver disponível. Além disso, um mesmo RDD não pode estar sob responsabilidade de mais de um *context* e uma troca entre contextos também não é permitida.

A criação de um RDD se dá por operações determinísticas de transformação ou ação. Transformações são operações que retornam, como resultado, um novo RDD. Esse tipo de operação pode ser vista como uma alteração em um RDD, tanto em seus dados como em suas características de particionamento, persistências e outros fatores. Já as ações são operações que retornam um valor final ao *driver* ou geram um *output* a um repositório externo. A Figura 2.4 exemplifica o funcionamento de transformações e ações.

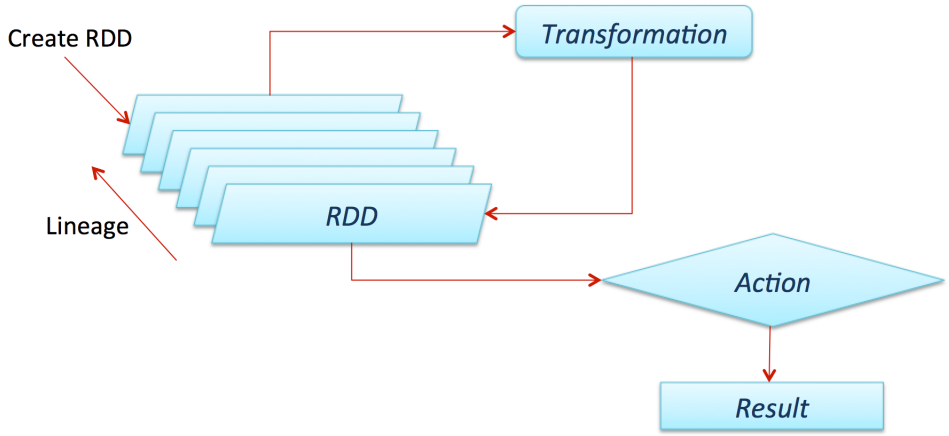


Figura 2.4 – Funcionamento das ações e transformações em RDDs no Spark (VISWANATH, 2016).

Um importante aspecto das operações em RDDs é o processamento baseado no modelo *lazy operation*: um *dataset* não é computado tão logo é definido, mas apenas quando um resultado final é requisitado. Por consequência, apenas operações do tipo ação geram processamento imediato. Já as transformações apenas definem novos RDDs no *context* da aplicação, poupando um procedimento computacional que pode não ser útil no momento de sua criação.

Os dados descritos por RDDs são provenientes de outra fonte de armazenamento seguro ou a partir de outros RDDs. Por ser um elemento *read-only*, um RDD não pode ser modificado depois de sua criação. Ao invés disso, uma operação em um *dataset* existente gera um novo RDD ou um resultado final, nos casos de transformação e ação respectivamente. Assim, aplicações com uma série de operações em RDDs geram grafos contendo o fluxo da aplicação, onde são armazenadas informações sobre as transformações sofridas por cada *dataset*. O fluxo,

também conceituado como a linha do tempo de cada RDD, é chamado de *lineage* e permite uma visualização de todo o processo de criação do *dataset*.

2.3.2.1 Jobs

Uma aplicação Spark possui uma estrutura de *tasks*, *stages* e *jobs*. Internamente, a chamada de uma ação em um RDD inicia um *job* que processa todas as partições do *dataset* até chegar ao resultado final. Um *job* tem seu *lineage* representado pelo escalonador de tarefas em uma estrutura de grafo acíclico dirigido (*Directed Acyclic Graph*, ou DAG). O escalonador ($DAGScheduler$) transforma as diferentes dependências do *dataset* em um plano físico de execução na elaboração do DAG e, então, distribui as tarefas entre os *executors* disponíveis.

O escalonador utiliza uma abordagem orientada a etapas, em que *stages* são criados para cada parte da execução de uma aplicação. A divisão da aplicação em etapas é geralmente usada para definir limites de execução paralela de tarefas. Enquanto uma partição pode ser processada sem necessidade de agrupamento, mantém-se o mesmo *stage*. Por conseguinte, um *stage* define uma série de tarefas independentes em partições de RDDs, de modo que o mesmo processo de um *stage* pode conter diferentes operações, mas deve ser executado em todas as partições. Quando há agrupamentos, novos *stages* são definidos.

A construção do DAG também depende da definição de dependências, que determina a relação entre cada partição dos *datasets* envolvidos. Uma relação assume um dos dois tipos: (a) *narrow* (1-1), em que cada partição de um RDD pai é usada por 1 RDD filho, no máximo; e (2) *wide* (1-N), em que múltiplos RDDs filhos podem depender de uma única partição do RDD pai. A Figura 2.5 ajuda a exemplificar os possíveis tipos de dependências.

Operações com dependências do tipo 1-1 geralmente não requerem comunicação com o *driver* ou com outros nós, podendo ser computadas de forma individual. Esse recurso evita procedimentos de comunicação entre elementos do ambiente computacional, tornando o processamento mais eficiente à medida em que os atrasos de comunicação são evitados. Um exemplo geral são as operações de mapeamento (*map*) e filtro (*filter*), as quais podem ser executadas de forma individual em cada partição (gerando apenas dependências 1-1) para que sejam agrupadas posteriormente. Teoricamente, a complexidade de aplicações que utilizam esse modelo de execução pode ser reduzida pela metade (KARAU; WARREN, 2017).

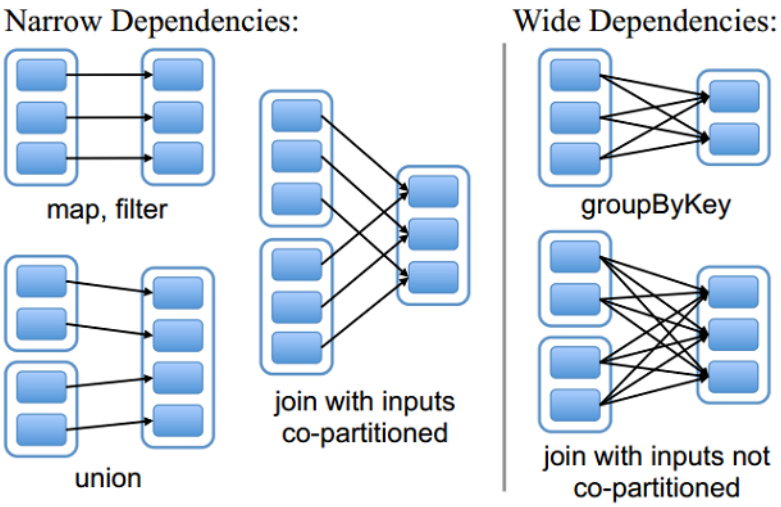


Figura 2.5 – Tipos de dependências em RDDs a partir de operações do Spark (ZAHARIA et al., 2012).

2.3.2.2 Persistência

A partir do processo de computação de um RDD – em uma ação –, todas as transformações são executadas para se chegar ao resultado final. Em caso de novas ações sobre um mesmo RDD, a computação deve ser realizada novamente mesmo que a operação seja a mesma já processada. Para evitar esse procedimento, pode-se persistir o conteúdo que já foi processado para um reaproveitamento em novas operações sobre o mesmo *dataset*. Esse recurso é essencial para diversas aplicações, sobretudo quando trabalha-se com algoritmos iterativos em que o reaproveitamento de dados é feito de forma constante.

O processo de persistência consiste no armazenamento de cada partição de um determinado RDD no seu respectivo nó computacional. Todos os nós que possuem partições de um RDD a ser persistido realizam o armazenamento do *dataset* de forma local. O meio de acesso da aplicação aos dados persistidos é o componente $BlockManager$, cuja função é mapear blocos de RDDs persistidos em memória ou disco. Segundo a documentação oficial da ferramenta (FOUNDATION, 2019a), a persistência pode auxiliar em ganhos de desempenho no processamento de ações futuras sobre o mesmo *dataset*.

Os métodos $persist()$ e $cache()$ são disponibilizados pelo Spark para realizar o armazenamento de dados de tal modo que não haja perda de dados ou metadados (i.e. *lineage*). A diferença entre os dois métodos é um detalhe sintático, já que o $cache()$ define o método de persistência com a opção *memory-only* como padrão. Sendo assim, o destino de armazenamento dos dados – através dos métodos de persistência – pode ser definido de três maneiras distintas:

memory-only, *disk-only* e *memory-and-disk*.

Na primeira opção, o RDD só poderá ser armazenado em memória principal, podendo ser persistido de forma natural ou serializado. Com serialização, o RDD ocupa um espaço menor mas necessita de um maior processamento para leitura e para escrita. Já com a opção *disk-only*, apenas o disco local é utilizado para a persistência, o que disponibiliza uma maior quantidade de espaço disponível mas causa um maior tempo de acesso. Há também a opção híbrida *memory-and-disk*, que busca armazenar uma quantidade máxima de dados em memória, levando-os ao disco quando o espaço torna-se limitado.

O Spark usa a política LRU (*Least Recently Used*) para remover RDDs antigos e liberar espaço para novas entradas quando o método de persistência *memory-and-disk* é usado. Com essa política, *datasets* mais antigos são considerados menos relevantes, pois tem um menor interesse das aplicações. Os RDDs liberados são processados novamente caso necessário. A política LRU é usada para o *swap* de dados da memória para o disco, quando a memória está cheia, e para a liberação completa do *dataset* quando o disco está cheio. A eficiência das aplicações depende de um controle otimizado dos dados armazenados, já que operações de escrita podem influenciar no fluxo de execução.

O espaço disponível em memória para persistência é definido com uma porcentagem do total de memória disponível no sistema. Apenas 75% de memória é alocada para o armazenamento de dados e execução de tarefas, sendo que o restante é reservado para metadados do Spark (como informações sobre RDDs e suas dependências). Dos 75% alocados, 50% é destinado para armazenamento de dados e os outros 50% é destinado a memória de execução (LASKOWSKI, 2019).

Uma alternativa aos meios de persistência via $cache()$ e $persist()$ é o salvamento de RDDs em *checkpoint*, via método $checkpoint()$. O *checkpoint* consiste no armazenamento de *datasets* em um repositório seguro, com uma diferença significativa dos outros métodos: o *lineage* é totalmente eliminado. Ou seja, o *checkpoint* realiza um truncamento do *lineage* de determinado RDD antes de persisti-lo, tornando-o um RDD raiz. A Seção 2.3.3.2 trata do método de *checkpoint* com maiores detalhes.

2.3.3 Tolerância a Falhas

O Spark foi projetado para oferecer uma técnica de tolerância a falhas com baixo custo e alta eficiência (KARAU; WARREN, 2017). Essa característica é possível pela abstração de

dados em memória definido por RDDs. Para auxiliar na recuperação de partições perdidas, há a noção de linha do tempo (*lineage*). Com o *lineage*, a base da recuperação de partições perdidas é a recomputação.

Porém, existem casos em que esse procedimento pode se tornar intrusivo, devido a fatores como complexidade e tamanho do *dataset* e das operações. Nesse sentido, o Spark também disponibiliza uma implementação da técnica de *checkpoint*, que é usada como uma forma de persistência segura de RDDs e, também, como uma solução para recuperações pós-falha.

2.3.3.1 Lineage

Além de armazenar dados, um RDD armazena metadados relacionados a sua construção (*lineage*, ou linha do tempo). Por ser *read-only*, um RDD não pode ser alterado sem que um novo seja criado. Nesse sentido, a linha do tempo é definida pelas transformações sofridas por um RDD desde sua fonte de dados original, além de suas relações com outros RDDs (dependências).

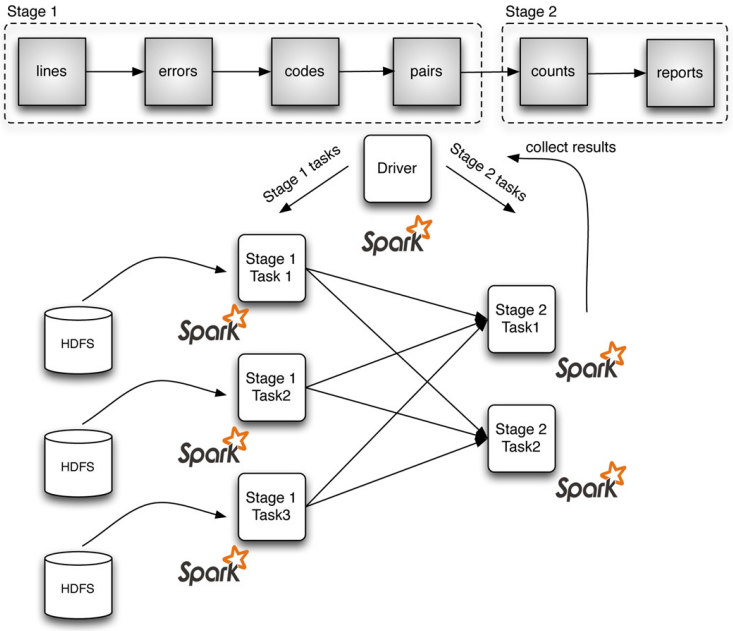
A Figura 2.6(b) exibe um exemplo de linha do tempo criada a partir das transformações do pseudo-código apresentado pela Figura 2.6(a). Cada operação gera um novo *dataset* com referência ao anterior. O RDD $errors$, por exemplo, consiste em uma dependência ao RDD $lines$ através da operação $filter$. Uma nova operação (map) em $errors$ gerou um novo *dataset*: $codes$. E, assim, todas as dependências são definidas até que uma ação seja requisitada. No exemplo da Figura 2.6, a ação consiste no método $collect$.

A característica que permite a construção de um RDD desde seu elemento raiz pode ser considerada a base da tolerância a falhas no Spark. Se uma partição é perdida, devido a algum tipo de falha, seu RDD possui a capacidade de reconstruir apenas a parte do *dataset* falha a partir de outras partições com estado estável. Assim, a recuperação de dados é possível mesmo sem a utilização de técnicas como replicação. Em comparação com outros métodos de abstrações de armazenamento em memória, os custos também se mantém baixos (ZAHARIA et al., 2012).

Em um evento de falha, as partições perdidas podem ser recuperadas através de um procedimento de recomputação, que consiste em resgatar informações de processamento de partições do mesmo RDD – que não tenham sido prejudicadas pela falha – e processá-las novamente. Em outras palavras, um RDD possui uma replicação de metadados distribuída em suas partições e, por conseguinte, a falta de uma partição não implica na perda do *dataset* como um todo. Sendo assim, informações armazenadas em outras partições auxiliam na reconstrução dos

```
1  lines = sc.textFile("hdfs://...")
2  errors = lines.filter(_.startsWith("ERROR"))
3  codes = errors.map(_.split("\t")(1))
4  pairs = codes.map(word => (word, 1))
5  counts = pairs.reduceByKey(_ + _)
6  reports = counts.map(kv => (dscr(kv._1), kv._2))
7  reports.collect.foreach(println)
```

(a) Exemplo de código para o filtro de mensagens de erro em um texto armazenado no HDFS.



(b) Representação do código apresentado em forma de DAG.

Figura 2.6 – Definição do grafo de execução de ações no Spark a partir de operações (adaptado de (INTERLANDI et al., 2015)).

dados perdidos.

O uso do *lineage* como técnica de tolerância a falhas requer que fontes de dados externas ao Spark também ofereçam características tolerantes a falhas. Esse requisito deve ser respeitado sobretudo em situações de falha em um RDD que seja diretamente dependente de um *dataset* externo. Nesse caso não há recomputação e os dados são requisitados dessa fonte, que também pode estar em estado falho. Por isso, recomenda-se que os dados de entrada e saída externas de RDDs estejam armazenados repositórios adequados, como o HDFS.

A eficiência da recuperação de dados por RDDs depende de alguns fatores. Um elemento essencial é a especificação de dependências do tipo *narrow* (ou 1-1) entre *datasets*. Uma quantidade maior desse tipo de dependência gera um processo mais rápido de recuperação, já que uma partição perdida pode ser reconstruída sem a necessidade de transferência de dados entre mais de um *worker*. Como uma dependência 1-1 indica que determinado RDD possui

apenas um predecessor, apenas este deve trabalhar no processo de recuperação. Ainda, esse tipo de dependência mantém os RDDs no mesmo nó, evitando transferências de dados pela rede. Sendo assim, o custo computacional de recomputação deve ser levado em consideração no procedimento de recuperação de *datasets* a partir da linha do tempo.

2.3.3.2 Checkpoint

A escolha do nível de armazenamento é importante para o desempenho do Spark, visto que os diferentes métodos possuem suas especificidades. Em geral, a opção de armazenamento em memória é mais eficiente que o disco, por seu método de acesso mais rápido aos dados. Porém, como o espaço em disco é mais barato, a opção por esse nível pode ser considerada. Nesse caso, deve-se avaliar o custo de reprocessamento do *dataset* em questão. Em muitos casos, a operação de leitura pode ser mais custosa que o processamento do RDD, inviabilizando o ganho de desempenho almejado pelo processo de persistência.

Nesse cenário, a técnica de *checkpoint* pode ser encontrada no Spark com o objetivo de persistir os RDDs, criando uma alternativa ao método *cache*. A diferença essencial do *checkpoint* para os outros métodos de persistência citados é o conteúdo salvo: em um *checkpoint*, apenas os dados do próprio RDD são salvos, de modo que suas dependências são totalmente eliminadas. Ou seja, o processo de *checkpoint* realiza uma etapa de truncamento do *lineage* para que o RDD torne-se um *dataset* sem dependências.

O processo de *checkpoint* é criado pelo desenvolvedor através do código de sua aplicação. A partir da indicação de salvamento de *checkpoint*, o Spark segue uma série de procedimentos até a finalização do processo. As etapas são descritas por quatro definições principais: inicialização, marcação de *checkpoint*, *checkpoint* em progresso e finalização. A etapa inicial é realizada a partir do comando $checkpoint()$. Porém, antes do uso de operações de *checkpoint*, deve-se estabelecer o diretório de salvamento do *dataset*.

Depois da inicialização, o RDD passa a ser da classe $RDDCheckpointData$, que contém os métodos e informações para realizar o processo. A classe, então, marca o RDD com o atributo $markedForCheckpoint$. Por se tratar de um *job* independente, o salvamento deve esperar que o término do *job* anterior (que processou o *dataset*) para ser iniciado. Na próxima etapa, quando o *job* de *checkpoint* é efetivamente iniciado, o sistema identifica o RDD a ser salvo, marca-o como $CheckpointingInProgress$ e recebe informações do local de armazenamento.

Ao finalizar o *job*, o RDD é marcado como $checkpointed$ e todas as suas dependências originais são eliminadas. A partir desse momento, uma única dependência é adicionada ao RDD, que consiste em uma referência à classe $CheckpointRDD$. Essa instância é usada para indicar o local de salvamento do *dataset* salvo e para gerenciar a criação das partições. A dependência gerada deve assumir uma de duas extensões: $ReliableCheckpointRDD$, no caso de armazenamento seguro em um sistema de arquivos externo, ou $LocalCheckpointRDD$ para o caso de armazenamento local.

Os *checkpoints* seguros preocupam-se exclusivamente com a tolerância a falhas dos dados persistidos. Nesse caso, apenas repositórios que garantam confiabilidade de dados são aceitos, podendo haver uma sobrecarga em relação aos procedimentos de segurança do sistema. Já o *checkpoint* local privilegia o desempenho do processo em detrimento da tolerância a falhas. Como consequência, os dados salvos podem ser irrecuperáveis em cenários de falha.

Em caso de falha com perda de alguma partição, o procedimento de recuperação segue o mesmo dos RDDs sem *checkpoints*. Porém, ao invés de realizar recomputações baseadas no *lineage* – que no *checkpoint* é eliminado –, a recuperação trata o RDD salvo ($CheckpointRDD$) como a fonte de dados original. A partir desse RDD, novos *datasets* podem ser criados normalmente. Em caso de falha, os novos *datasets* terão suas partições recomputadas de acordo com o *lineage*, mas tendo o RDD salvo em *checkpoint* como o elemento raiz, assim diminuindo a quantidade de transformações necessárias mas aumentando a quantidade de operações de leitura.

O uso de *checkpoint* pode ser útil para diversos cenários, porém destaca-se os casos em que o processo de computação de um RDD é longo, devido à grande quantidade de dependências e RDDs predecessores. Nesse cenário, a alternativa mais eficiente pode ser o salvamento do *dataset* em *checkpoint*. Assim, todo o processo descrito pelo *lineage* é cortado e o *dataset* pode trabalhar com novas operações sem o custo das dependências anteriores. Além disso, o *checkpoint* tem a vantagem de não depender da execução do $SparkContext$ para se manter disponível, ao contrário dos métodos de *cache* e *persist*, já que seus dados podem ser recuperados mesmo após a interrupção do*context*. Esse recurso pode ser utilizado em situações de reexecução de uma mesma aplicação, em que o *dataset* tem a oportunidade de ser obtido a partir de seu *checkpoint*, ao invés de haver uma recomputação.

2.3.4 Metrics System

Uma alternativa para o monitoramento de aplicações no Spark se dá através de seu sistema de métricas chamado $MetricsSystem$. Nesse sistema, há uma divisão dos principais elementos do *framework* em $sources$. Nesse caso, elementos como o *driver*, o *context* e *executor* possuem seus próprios $sources$. Um $source$ possui a tarefa de coletar dados de utilização do elemento correspondente e expor esses dados de acordo com a necessidade do usuário.

O $MetricsSystem$ do Spark oferece a liberdade de consumo dos dados coletados por $sources$ através de elementos externos chamados $sinks$. Para cada elemento monitorado pode-se definir um ou mais $sinks$, que definem o cliente para qual os dados serão reportados. O cliente é geralmente definido por um arquivo de saída, configurado previamente. Os resultados de monitoramentos via $sources$ são salvos nesse arquivo assim que coletados.

O próprio Spark também pode consumir informações provenientes do $MetricsSystem$ em tempo de execução. Essa funcionalidade é encontrada na interface *web* do Spark, executada por padrão na porta 4040 do *host* desejado (quando o $StandaloneCluster$ é utilizado como $ClusterManager$). Além disso, as métricas são exportadas via JSON pela API REST do sistema, o que possibilita uma análise de recursos via requisições HTTP.

3 ARQUITETURA DE CONFIGURAÇÃO DINÂMICA

O *checkpoint* apresenta-se como uma importante técnica no contexto da tolerância a falhas. Porém, configurá-lo de forma eficiente é um grande desafio, uma vez que diversos fatores podem interferir no seu comportamento. Ainda que sobrecargas sejam aspectos intrínsecos em técnicas de tolerância a falhas para que a eficiência de recuperação seja garantida (PULLUM, 2001), é necessário buscar um equilíbrio entre desempenho e confiabilidade (NETZER; XU, 1993). Sob configurações inapropriadas em implementações do *checkpoint*, o sistema e as aplicações sofrem com problemas de confiabilidade e desempenho. Nesse caso, sua característica de tolerância a falhas perde o propósito essencial.

Para que o equilíbrio entre desempenho e confiabilidade seja definido, é importante haver um monitoramento das atividades do ambiente em que se trabalha. Desta forma, é possível identificar cenários favoráveis para o uso ou para a contenção das operações de *checkpoint*. Isto é, *checkpoints* devem ser considerados à medida que a confiabilidade do sistema tende a diminuir, ou evitados em perspectivas de baixo desempenho. Porém, dadas as características manuais e estáticas presentes no Hadoop e no Spark, a adaptação da técnica de *checkpoint* em tempo de execução é infactível.

Sendo assim, este capítulo apresenta a proposta de uma arquitetura para configuração dinâmica para atributos de *checkpoint*, denominada *Dynamic Configuration Architecture* (DCA), com implementações para os *frameworks* Hadoop e Spark. O objetivo da arquitetura é tornar a configuração de atributos relacionados à técnica mais eficiente. Com atributos adaptados dinamicamente, espera-se que a técnica de *checkpoint* tenha suas propriedades de tolerância a falhas melhor exploradas, ao mesmo tempo em que sua intrusividade seja controlada, a partir da observação do sistema.

3.1 Arquitetura Geral

A DCA possui dois elementos essenciais: o *monitor* e o *coordinator*. Além disso, implementações para o Hadoop e para o Spark – descritas nas Seções 3.4 e 3.5, respectivamente – foram definidas para que adaptações no funcionamento de seus *checkpoints* sejam possíveis. A arquitetura proposta, bem como seus elementos e a comunicação entre eles, é mostrada de forma geral na Figura 3.1.

O funcionamento da arquitetura pode ser sumarizado pelas comunicações entre os ele-

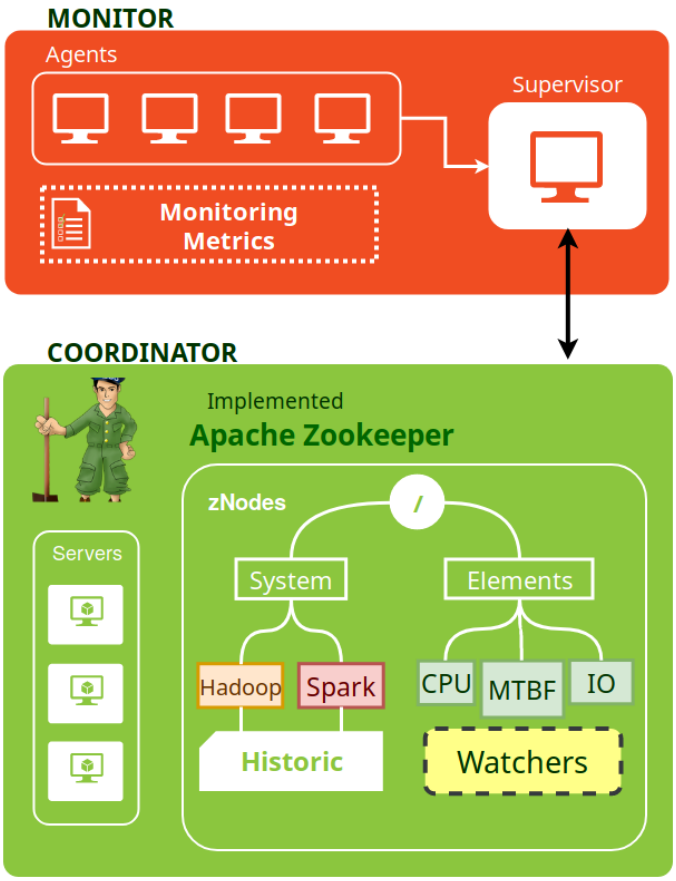


Figura 3.1 – Arquitetura de configuração dinâmica (DCA) para atributos de *checkpoints*.

mentos. O *monitor* possui comunicação com o *coordinator*, a fim de enviar alertas sobre mudanças de contexto observadas. Uma mudança de contexto indica quando o comportamento de um elemento do sistema sofreu algum tipo de alteração desde o último momento em que foi monitorado. Nesse caso, adaptações dos atributos de configuração ao novo contexto podem ser úteis. Dessa forma, a arquitetura oferece uma visão do comportamento do sistema em tempo real para que adaptações sejam possíveis, com um foco voltado às propriedades adequadas para determinados momentos.

O *coordinator* é implementado para gerar alertas às implementações nos *frameworks* e mantém uma comunicação em dois sentidos com o *monitor*. Quando há uma alteração em sua composição, devido a mudanças de contexto recebidas do *monitor*, um mecanismo de aviso é executado. Qualquer implementação que esteja observando o *coordinator* também é notificada. Por consequência, o *coordinator* age como uma interface entre os recursos monitorados e as implementações usadas. Esse recurso pode ser usado por outros *frameworks* e sistemas, apenas ligando-os com o *coordinator* para que possam receber alertas a respeito do andamento do sistema. Os detalhes de cada elemento e das implementações feitas são descritos a seguir.

3.2 Gerenciamento

Visando um armazenamento seguro de informações de configuração, o *coordinator* foi usado como um repositório central para o gerenciamento de atributos de configuração, a partir do *framework* Apache Zookeeper: um projeto *open source* com funcionalidades para facilitar a coordenação de sistemas distribuídos (HUNT et al., 2010). A arquitetura do Zookeeper é formada por servidores que realizam as operações requisitadas por clientes em um $namespace$ compartilhado cuja estrutura é baseada em uma árvore. Seus nós ($zNodes$) são usados para guardar informações de configuração: dados de até 1MB.

Os atributos relacionados aos fatores de monitoramento do *monitor* possuem nós específicos. Além disso, há um nó correspondente ao Hadoop e outro ao Spark, para facilitar uma obervação geral dos *frameworks* ao *coordinator*. Uma importante função do Zookeeper em sua atuação como *coordinator* é a configuração de mecanismos de alerta (*watchers*), que geram notificações para cada modificação em um $zNode$. Ou seja, a partir de qualquer alteração em um atributo de configuração, todo sistema que possua um *watcher* ao $zNode$ em questão é notificado.

3.2.1 Histórico

Estabelecer uma análise preditiva de comportamento dos elementos de um sistema é um grande desafio quando há pouca ou nenhuma informação disponível de forma antecipada (i.e. *a-priori*). Essa limitação compromete a otimização de tarefas e de ferramentas que, em razão disso, dependem da especificação estática e não automatizada de atributos – como é o caso do procedimento de *checkpoint* nos *frameworks* Hadoop e Spark. Nesses casos, a união de configurações estáticas com a falta de informação *a-priori* condiciona desenvolvedores a implantarem uma abordagem de "tentativa e erro" para adaptar o sistema de forma eficiente. Nem sempre, porém, obtém-se sucesso, especialmente quando a heterogeneidade do sistema é um aspecto relevante.

Monitorar recursos do sistema em tempo real pode ser uma solução eficiente para esse problema. Contudo, ainda que esse tipo de monitoramento consiga determinar a situação momentânea de fatores de observação, uma única amostra pode ser insuficiente para que uma adaptação ótima seja alcançada. Os ambientes computacionais – sobretudo os sistemas distribuídos – sofrem diversos tipos de perturbação durante o funcionamento de seus serviços. Condições

inesperadas como instabilidade de rede e mal funcionamento de componentes, além da diversidade de aplicações, cargas de dados e/ou características (físicas ou lógicas) de componentes, comprometem análises preditivas de comportamento quando a amostragem é limitada.

No contexto da DCA, uma análise mais robusta dos elementos monitorados torna-se essencial para garantir a tangibilidade do dinamismo pretendido. Diversas soluções de estimativa são utilizadas pela literatura com observações baseadas em *job profilling* (POLO et al., 2011) (MASHAYEKHY et al., 2014). Contudo, a dependência do desenvolvedor para a criação de estimativas de comportamento remete a um comportamento estático de observação.

Para que a DCA utilize um meio dinâmico de estimativa, o *coordinator* foi projetado para funcionar como um repositório de metadados do sistema em um modelo de histórico. À medida que novas informações são coletadas pelo *monitor*, informações antigas deixam de ser descartadas e passam a ser consideradas em cálculos de predição. Assim, o histórico tem o potencial de identificar anomalias de funcionamento quando um evento incomum e isolado é experimentado para mitigar seu impacto nas análises posteriores. Do mesmo modo, quando o sistema realmente sofre uma mudança notável (não configurada como um evento isolado), o histórico também pode auxiliar as análises através da identificação desse novo comportamento.

Para cada elemento monitorado, um $zNode$ pode ser definido com um caminho específico. Nessa estrutura, um elemento possui três nodos descendentes: um nodo para armazenar o último elemento adicionado ($last$), outro para armazenar os valores obtidos por observações ($values$) e, por fim, outro para o armazenamento de análises estatísticas ($analysis$). A Figura 3.2 auxilia na compreensão da arquitetura de organização dos $zNodes$ implementada pelo histórico do *coordinator*.

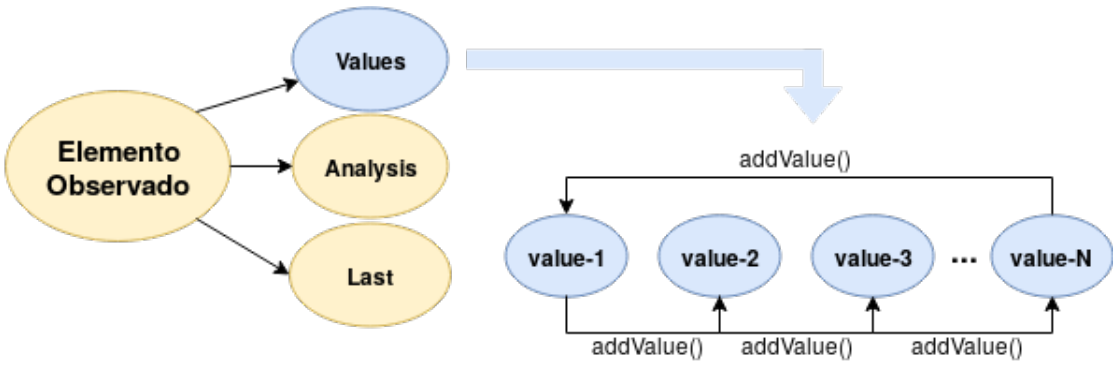


Figura 3.2 – Arquitetura de histórico do *coordinator* com uma janela de N entradas.

Para que se obtenha uma visão a respeito do comportamento de um elemento, é necessário armazenar uma quantidade suficiente de informações. Por isso, além do atributo de

configuração atual, o *coordinator* mantém todos os atributos já utilizados, para então, criar a concepção de histórico. Na árvore de $zNodes$, o nodo $values$ é composto por um número máximo de N descendentes, que corresponde à janela de observação. Logo, o histórico armazena os últimos N valores observados. A atualização é baseada no conceito de *Round Robin Database* (RRD), sendo que os valores são atualizados nos descendentes de $value$ a partir de 1 até N e, após N, a partir de 1 novamente. O nodo $last$ é usado para observação externa, de modo que qualquer ferramenta tenha a noção de quando um elemento tem seus dados atualizados por meio de alertas do Zookeeper.

3.3 Monitoramento

O monitoramento de recursos através do elemento *monitor* é uma etapa essencial da configuração dinâmica dos atributos de *checkpoints*. O objetivo deste módulo é analisar a situação do sistema de forma global e/ou local, para que seja possível realizar uma quantificação da utilização de seus recursos. A partir disso, o mecanismo tem a possibilidade de identificar possíveis cenários para a adaptação de atributos de *checkpoint*, tanto do Hadoop como do Spark, através de métricas de monitoramento. Assim, o *monitor* oferece visões correspondentes ao contexto do sistema para manter a arquitetura dinâmica atualizada sempre que possível.

Assim como a grande parcela das ferramentas de monitoramento de recursos (ACETO et al., 2013), a arquitetura do módulo *monitor* usa um modelo agente-servidor. Nesse caso, 1 ou N agentes (*agents*) são executados com base na comunicação de um único servidor (*supervisor*). A ideia é manter uma coordenação de monitoramento centralizada e simples, capaz de gerar informações através de *agents* e administrá-las no *supervisor*. Os elementos do *monitor* são explorados a seguir.

3.3.1 Agents

Um *agent* é executado para cada elemento do ambiente computacional que tenha a necessidade de ser monitorado, de modo que informações sobre seus recursos são coletadas em tempo real, periodicamente ou sob demanda. Quando uma mudança de contexto significativa é detectada, o agente deve mandar uma mensagem de *report* ao servidor. Nesse sentido, há dois tipos de *agents*. Os *external agents* trabalham diretamente junto ao módulo de monitoramento, independentemente dos sistemas-alvo (neste caso, Hadoop e Spark). Já os *internal agents* são

implementados nos sistemas usados e funcionam de acordo com a demanda necessária, com o objetivo de coletar informações que *external agents* não são capazes de identificar do lado de fora das ferramentas. Ambos os *agents* são descritos a seguir.

3.3.1.1 External Agents

Os *external agents* coletam dados de maneira periódica em nodos específicos. Os recursos que podem ter informações coletadas são: (a) CPU, (b) RAM e (c) *I/O rate*. A coleta é feita periodicamente, sendo que este período deve ser especificado de acordo com a necessidade de monitoramento. Trabalhos recentes mostram que a intrusividade do monitoramento periódico é baixa mesmo quando um período frequente (em torno de uma coleta a cada 10 segundos) é utilizado (CARDOSO; BARCELOS, 2018a) (CARDOSO; BARCELOS, 2018b).

Uma mensagem de *report* é enviada ao *supervisor* assim que um dos elementos monitorados tem uma alternância de comportamento significativa, desde a última observação. Ou seja, o *external agent* obtém a taxa de utilização atual do elemento e compara com o último valor obtido antes da observação atual. Em havendo uma diferença maior do que o tolerável, considera-se que existe uma mudança de contexo. Isto é, quando uma mudança de contexto é observada, entende-se que as novas coletas devem ser observadas pelo *supervisor* para evitar que os atributos atuais de *checkpoint* não estejam defasados pelo novo estado do sistema. Os níveis de tolerância para *external agents* são especificados por métricas de monitoramento, definidas pelo *supervisor*.

3.3.1.2 Internal Agents

Nos *internal agents*, métricas de monitoramento específicas para cada sistema são definidas para a coleta de atributos internos. A implementação deste tipo de agente deve-se ao fato de que *external agents* são incapazes de coletar estatísticas internas às ferramentas, tais como os custos em tempo de execução de determinados procedimentos.

Foram implementados dois *internal agents* no Apache Hadoop, os quais controlam: (a) o custo para o salvamento de *checkpoints*, medido pelo tempo de execução do procedimento; e (b) a presença de falhas no HDFS, essencialmente no $NameNode$. O *internal agent* foi implantado no elemento $SecondaryNameNode$. Assim, quando um *checkpoint* é salvo, o Hadoop envia uma mensagem de *report* ao Zookeeper – através do *internal agent* –, contendo o tempo em segundos. Para o controle de falha, o procedimento é similar. Assim que uma falha é detectada,

o *internal agent* comunica-se com o *coordinator* indicando o tempo decorrido desde a última falha observada (de acordo com os *timestamps* das falhas). No caso da primeira falha, o tempo é calculado a partir do início de execução do *framework*.

No Spark, os *internal agents* realizam a observação sob demanda dos *datasets* usados assim que um *job* é submetido. Os *agents* são executados para cada aplicação de forma individual. Ou seja, cada $SparkContext$ possui seu próprio *internal agent*. O objetivo desse agente é indicar o desempenho de um *job* através do seu tempo de execução (em segundos) e da complexidade do *dataset* envolvido. O número de partições $part$ foi usado como métrica para definir a complexidade do RDD. Assim, o desempenho do *job* ($jobTime$) é dado em $segundos/part$. A escolha pelo número de partições se dá pela relação 1-1 entre blocos do HDFS e partições de RDDs (LASKOWSKI, 2019), facilitando a análise de *datasets* salvos em *checkpoint*.

3.3.2 Supervisor

O *supervisor* é o elemento mestre do monitoramento e mantém uma comunicação passiva com os *agents* (tanto *internal* e *external agents*). Seu funcionamento consiste em esperar por mensagens de *report* enviadas pelos *agents* para definir novos atributos a serem enviados ao *coordinator*. Em outras palavras, o *supervisor* atua como um elemento de monitoramento central que coordena os *agents* e os dados recebidos de seus *reports*. Quando uma mensagem é recebida, o *supervisor* deve calcular adaptações de atributos – neste trabalho, o período entre *checkpoints* do Hadoop.

Os *reports* de *external agents* são recebidos a partir de uma comunicação via protocolo UDP, tão logo um agente percebe uma mudança de contexto. Por outro lado, *reports* de *internal agents* são recebidos via *watchers* do Zookeeper, uma vez que o *supervisor* mantém uma observação aos nodos $last$ de $zNodes$ que guardam informações sobre elementos monitorados.

O módulo de monitoramento requer o uso de métricas para que a DCA possa identificar mudanças de contexto. Também, métricas precisam ser especificadas para a tomada de decisões sobre mudanças nos atributos de *checkpoint*. Nesse sentido, a arquitetura proposta admite a especificação de métricas de duas maneiras. A abordagem estática consiste em um arquivo com valores sobre cada elemento observado. Nesse caso, cada elemento tem um valor associado que consiste na diferença máxima tolerada entre duas observações. Assim, deve-se estabelecer *thresholds* estáticos para que mudanças de contexto sejam definidas.

Outra métrica de monitoramento é a escolha dinâmica de atributos baseada em aproxi-

mações para o período ideal entre *checkpoints*, no Hadoop; e, baseada em políticas dinâmicas, no Spark. Essa abordagem é extremamente útil quando métricas estáticas são difíceis de serem especificadas, de modo que a característica dinâmica da arquitetura é melhor explorada. Cálculos automatizados definem novos atributos de acordo com o comportamento atual do sistema.

Neste trabalho, o *supervisor* implementa duas aproximações para o cálculo de períodos entre *checkpoints* no Hadoop, propostas por trabalhos relacionados que são descritos na Seção 3.4.2. Já as políticas dinâmicas do Spark são propostas pela DCA nas Seções 3.5.3 e 3.5.4.

Além de implementar métricas, o *supervisor* atua como elemento central do histórico implementado no *coordinator*. A média dos dados de *internal agents* – armazenados pelos filhos do $zNode\ values$ – é calculada pelo *supervisor*. Após esse cálculo, a média é enviada de volta ao $zNode\ analysis$. Para o cálculo da média, o *supervisor* conta com o auxílio do histórico no arranjo dos nodos em $values$. Os dados são organizados com base na política RRD, com uma janela de observações de tamanho fixo. Assim, a média calculada tem como limiar as últimas N observações feitas.

Nos casos de análises com dados provenientes de *external agents*, apenas os dois últimos valores observados são levados em conta. Assim, o histórico do elemento não é considerado e a avaliação compreende apenas uma análise imediata. Esse tipo de observação é útil quando há necessidade de verificar-se unicamente o estado atual de um elemento, ou para casos em que uma mudança de contexto é determinante mesmo quando acontece de forma isolada.

3.4 Implementação no Hadoop

O processo de *checkpoint* pode se tornar um fator crítico para o desempenho do Apache Hadoop. A escolha do intervalo entre *checkpoints* implica em diferentes comportamentos da recuperação (em caso de falha) e do andamento de aplicações. Intervalos curtos tendem a oferecer mais confiabilidade, mas adicionam uma quantidade considerável de operações de salvamento. Por outro lado, intervalos maiores evitam essa sobrecarga mas comprometem a recuperação com imagens de *checkpoint* defasadas. Por isso, a implementação da DCA no HDFS tem como objetivo a adaptação em tempo real – sem a necessidade de interromper o Hadoop e seus serviços – do período entre *checkpoints*.

No Hadoop, a técnica de *checkpoint* é implementada pelo SNN. Portanto, as mudanças realizadas para atender aos propósitos da DCA foram realizadas nesse elemento do HDFS. Nesse sentido, duas classes foram consideradas o núcleo do *checkpoint* no SNN: a classe

$CheckpointConf$ mantém dados de configuração do HDFS armazenadas em memória, para que a classe $Checkpointer$ os utilize enquanto lidera todo o procedimento de *checkpoint*.

3.4.1 Funcionamento

No mecanismo proposto neste trabalho, foram realizadas alterações no tratamento do período entre *checkpoints* na classe `Checkpointer`. Essas alterações incluem um elemento de comunicação entre o HDFS e o *coordinator*. Também foi implementado um mecanismo para tratamento de alertas, de modo que qualquer mudança de contexto identificada pela arquitetura seja refletida no $Checkpointer$. Assim, quando um cálculo sobre o novo período entre *checkpoints* é feito, o $Checkpointer$ reinicia o loop de *checkpoint* para assegurar-se de que o tempo de espera decorrido até o momento está de acordo com a nova configuração. Um novo procedimento de *checkpoint* é imediatamente iniciado caso necessário. Se não, a classe permanece esperando pelo tempo necessário, considerando o tempo já esperado e o novo período definido.

Em sua inicialização, o $Checkpointer$ confere a metodologia de configuração a ser usada. Isto é, a implementação da DCA não exclui a possibilidade de uso do mecanismo de configuração estática. A diferença consiste na comunicação com o *coordinator*, quando a abordagem dinâmica é usada. No caso estático, a origem desse atributo continua sendo os arquivos de configuração. Já no dinâmico, o período entre *checkpoints* é recuperado do *coordinator*. O Algoritmo 1 demonstra o método $get\_current\_period()$ – que obtém o período entre *checkpoints* –, além das modificações no *loop* responsável pelo procedimento de *checkpoint*.

É necessário ressaltar que o procedimento de salvamento de *checkpoints* propriamente dito não foi alterado, de modo que a abordagem dinâmica refere-se à dinamicidade do atributo de configuração referente ao intervalo entre *checkpoints*.

Após a inicialização do $checkpoint\_loop$, a única diferença entre as abordagens dinâmica e estática é possibilidade de modificações em tempo real no período entre *checkpoints* oferecido pela DCA. Essa diferença resume-se ao método de espera entre dois *checkpoints*. A abordagem estática usa o método $sleep()$ durante a espera, sendo que este é um método que não pode ser interrompido antes de atingir o tempo de espera definido.

Por outro lado, a abordagem dinâmica usa o método $wait()$ para o controle de tempo entre dois salvamentos de *checkpoint*. O funcionamento do $wait()$ difere do $sleep()$ na possibilidade de interrupção, em tempo de execução, antes do tempo de espera definido. A interrupção é feita através de uma notificação $wakeUp()$, que é executada no elemento de comunicação

Algoritmo 1: MODIFICAÇÕES EM MÉTODOS DO HDFS PARA SUPORTE A DCA

```
1  Function get_current_period():
2      if dynamic_configuration() then
3          period = get_from_zk();
4      else
5          period = get_from_conf();
6      end
7  end
8
9  Function checkpoint_loop():
10     last_checkpoint = 0;
11     while should_run do
12         period = get_current_period();
13         sleep_time = period − (time.now() − last_checkpoint);
14         if sleep_time > 0 then
15             if dynamic_configuration() then
16                 wait(sleep_time);
17             else
18                 sleep(sleep_time);
19             end
20         else
21             doCheckpoint();
22             last_checkpoint = time.now();
23         end
24     end
25 end
```

HDFS-*coordinator* toda vez que o $zNode$ do período entre *checkpoints* é modificado.

Além de esperar por modificações no período entre *checkpoints*, o elemento de comunicação HDFS-*coordinator*, implementado no $Secondary\ NameNode$, também define a execução de um *internal agent*. O papel desse agente no HDFS é atualizar o *coordinator* da DCA com estatísticas de uso sobre os salvamentos de *checkpoint*. Neste caso, o agente envia dados sobre o custo do salvamento de um *checkpoint*. O custo é calculado como o tempo necessário para que o $Checkpointer$ efetue o salvamento de um *checkpoint* no método $doCheckpoint()$ do Algoritmo 1. Além disso, o agente também coleta e envia a taxa de falhas observada no HDFS.

3.4.2 Métricas

A busca por períodos ideais entre *checkpoints* é um dos maiores desafios para a técnica de CR, essencialmente no Hadoop. Uma vez que *hardwares*, *softwares* e ambientes distribuídos podem assumir diferentes configurações – e, consequentemente, diferentes comportamentos –

a definição de um atributo global é uma tarefa difícil. Ademais, *frameworks* como o Hadoop bloqueiam adaptações em tempo real para esse atributo.

De modo a definir o período entre *checkpoints* de forma ideal, diversos estudos foram propostos, precisando-se estimar custos de salvamento e taxas de falha. Neste sentido, a proposta da DCA explora dois trabalhos com grandes contribuições no âmbito de definições eficientes para o período de *checkpoints*. Primeiramente, Young (YOUNG, 1974) propôs uma aproximação que apresentou uma minimização no tempo gasto para salvar *checkpoints*. Posteriormente, Daly (DALY, 2006) aprimorou o trabalho de Young a partir da criação de uma estimativa *high-order*.

Neste trabalho, as fórmulas desenvolvidas por Young e Daly foram indicadas como base para criar métricas de adaptação em tempo real para o período entre *checkpoints* do Hadoop. Uma métrica foi criada para cada trabalho citado, através de implementações no *supervisor* da DCA. Todos os dados coletados pelo *supervisor* e pelos agentes (internos e externos) auxiliaram na avaliação de novos valores pelas fórmulas. Esta seção apresenta os detalhes da implementação das fórmulas de Young e Daly no contexto da arquitetura de configuração dinâmica.

3.4.2.1 Aproximação de Young

A aproximação proposta por Young (YOUNG, 1974) foi desenvolvida com o intuito de reduzir o tempo gasto para realizar salvamentos em *checkpoints*. O cenário alvo do estudo consiste em sistemas com probabilidade de falhas baseada em uma distribuição aleatória. Baseado no tempo médio entre falhas do sistema (M) e o custo computacional de *checkpoint* (C), a fórmula de Young Δ_{young} é descrita na Equação 3.1.

$$\Delta_{young} = \sqrt{2 * C * M} \tag{3.1}$$

A fórmula de Young apresentou resultados satisfatórios em validações conduzidas na literatura. Os autores em (EL-SAYED; SCHROEDER, 2014) mostraram que a aplicação da fórmula habilitou uma redução da quantidade de processamento gasto, a partir da diminuição da carga de *checkpoint* a cada salvamento. Apesar de ser uma fórmula simples, com suposições que geralmente não refletem o contexto dos sistemas computacionais atuais, o desempenho oferecido pela simplicidade de implementação pode garantir bons resultados.

Contudo, a aplicação desta fórmula depende – de forma substancial – de um conhecimento prévio a respeito do tempo médio para que uma falha ocorra no sistema. Do mesmo

modo, o conhecimento sobre o custo do salvamento de *checkpoints* é essencial neste caso. Mas na prática, a definição desses atributos é dificultada por fatores como a variação de comportamento dos sistemas, os eventuais fenômenos externos, ou mesmo as diferentes cargas de dados usadas por aplicações.

De forma a corrigir esse tipo de problema, a implementação da fórmula de Young na DCA leva em consideração o tempo médio entre falhas e o custo computacional de *checkpoints* calculados pelo mecanismo de histórico no *coordinator*. Ou seja, há uma estimativa destes valores a partir de registros anteriores que é dinamicamente alterada conforme novas observações são feitas. De acordo com os novos dados, novos (e mais atualizados) valores são calculados pela fórmula.

3.4.2.2 Aproximação de Daly

O trabalho apresentado por Daly (DALY, 2006) descreve uma extensão da proposta de Young. A partir da implementação de um modelo *first-order* idêntico ao modelo de Young (DALY, 2003), os autores encontraram uma lacuna que limita a eficiência do primeiro modelo, que desconsidera o tempo necessário para a recuperação pós-falha. Essa lacuna é relevante, já que a etapa de recuperação pode ser longa o suficiente para apresentar uma falha. A solução encontrada para esse problema resultou na adição de apenas um único fator à equação: o próprio tempo de recuperação R. A Equação 3.2 descreve a fórmula *first-order* de Daly, que é semelhante à Equação 3.1. Ambas as fórmulas convergem para um mesmo resultado quando o tempo de recuperação tende a zero.

$$\Delta_{F-Daly} = \sqrt{2 * C * (M + R)} \tag{3.2}$$

O modelo de *first-order* de Daly apresenta duas asserções problemáticas quando o nível de confiabilidade do sistema é pequeno (i.e. quando M tende a ser menor que $\Delta + C$). As Equações 3.1 e 3.2 assumem que falhas jamais ocorrem durante uma operação de *checkpoint*. Essa afirmativa é considerada equivocada, de modo que muitos autores já consideraram a possibilidade da ocorrência de falhas durante salvamentos de *checkpoint* (EL-SAYED; SCHROEDER, 2014). Tratando-se dos sistemas de processamento intensivo de dados, o procedimento de salvamento de um *checkpoint* pode ser de longa duração. Assim, as chances de uma falha ocorrer nesse momento crescem à medida em que o custo de salvamento é maior.

Por esta razão, o trabalho em (DALY, 2006) traz um modelo (*high-order*) proposto como solução para as suposições problemáticas de seu primeiro modelo. A Equação 3.3 mostra a solução, que é apontada como uma aproximação mais completa por considerar um cenário mais próximo de ambientes de produção. Neste modelo, o tempo de recuperação R foi novamente removido da fórmula, visto que sua contribuição ao cálculo do período entre *checkpoints* foi considerada irrelevante. Porém, a solução de perturbação P da Equação 3.4 foi adicionada para um melhor controle de erros de precisão.

$$\Delta_{Daly} = \begin{cases} (\sqrt{2 * C * M}) * P - C & \text{when} \quad C < M \\ M & \text{when} \quad C \geq M \end{cases} \tag{3.3}$$

$$P = 1 + \frac{1}{3}\left(\frac{C}{2 * M}\right)^{\frac{1}{2}} + \frac{1}{9}\left(\frac{C}{2 * M}\right) \tag{3.4}$$

Os resultados apresentados mostraram um nível baixo de erro relativo obtido pela aproximação (DALY, 2006). Desta forma, a implementação da fórmula de Daly na DCA teve como foco a solução *high-order* devido ao seu nível mais apurado e seu potencial para obtenção de resultados mais precisos.

3.4.3 Falhas no NameNode

O $NameNode$ é o elemento central da arquitetura do HDFS. Por isso, um evento de falha neste elemento pode comprometer a disponibilidade dos dados e prejudicar o andamento das aplicações. Nas versões iniciais do Hadoop, o NN era considerado um ponto único de falha (SPOF - *single point of failure*). Já nas versões 2.x, o atributo de alta disponibilidade (*High Availability*) auxilia a recuperação do NN a partir de *backups* em tempo real, mas acrescenta operações extras ao contexto do Hadoop. Porém, falhas transientes no $NameNode$ podem ser reparadas apenas com seu reinício, evitando-se operações adicionais.

Reiniciar o NN durante o andamento de uma aplicação pode ser fatal, dado que uma interação mal sucedida entre a aplicação e o HDFS gera erros de execução. Nas etapas de início da execução, realizam-se a alocação de recursos e a preparação da plataforma. Neste momento, ainda que a aplicação não esteja em execução, o HDFS é usado para armazenar informações sobre a aplicação. Essas informações incluem o arquivo executável (JAR), configurações e informações sobre arquivos de entrada e saída (WHITE, 2015). A quantidade de tarefas criadas para a execução é baseada no número de divisões do arquivo de entrada. Para consultar essa

informação, um novo acesso é feito ao HDFS. Logo após, a execução inicia e as interações com o HDFS acontecem de acordo com a demanda da aplicação.

Para criar os cenários de falha transiente com recuperação imediata, mecanismos de tolerância a exceções foram definidos em cada interação do $NameNode$ com outros elementos. Para isso, métodos da classe de tradução do protocolo usado por clientes para comunicação com o $NameNode$[1] foram alterados. Os métodos: $addBlock$, $complete$, $create$, $delete$ e $getFileInfo$ foram modificados, uma vez que todos são responsáveis pela execução de comandos no HDFS.

As alterações realizadas nos métodos citados incluem um número máximo de tentativas de comunicação a serem feitas, antes de detectar o $NameNode$ como falho, além de um tempo de espera entre as tentativas. Essa espera é fundamental para que a aplicação não seja encerrada antes de uma possível recuperação, quando há falha. Por outro lado, as alterações não comprometem a execução das aplicações quando nenhuma falha é detectada.

3.5 Implementação no Spark

O funcionamento do *checkpoint* no Spark pode se tornar uma tarefa difícil e custosa, uma vez que o procedimento é manual e não possui nenhuma política de funcionamento que permita uma automatização. O *checkpoint* do Spark introduz operações de *I/O* ao ambiente de execução e é coordenado pelo desenvolvedor através do código, a partir do controle de quais *datasets* devem ser salvos, além do momento em que esses procedimentos acontecem. Como um mesmo sistema computacional pode apresentar um comportamento heterogêneo, devido aos diferentes processos a que é submetido, tomar decisões sobre o impacto da persistência de dados no desempenho do Spark é um grande desafio (DUAN et al., 2016), sobretudo quando usa-se *checkpoint*.

Para que decisões eficientes sejam selecionadas, o desenvolvedor deve possuir um conhecimento apurado do comportamento do sistema e da aplicação. Por isso, escolhas eficientes sobre o momento que um *dataset* deve ser salvo em *checkpoint*, ou sobre qual *dataset* é o mais apropriado para ser salvo, são difíceis de serem tomadas. Além disso, não há possibilidade de realizar qualquer adaptação sobre uma escolha de *checkpoint* em tempo de execução, dado que as escolhas se dão via código. Essa característica pode causar limitações de desempenho à medida em que decisões não otimizadas são tomadas. Ainda, escolhas erradas prejudicam a

[1] Classe `ClientNamenodeProtocolTranslatorPB`

característica tolerante a falhas da técnica, uma vez que a torna intrusiva sem que o custo de recuperação dos *datasets* seja reduzido.

3.5.1 Funcionamento

Para contornar os problemas vistos, este trabalho apresenta uma alternativa dinâmica para o *checkpoint* do Spark baseada em dois atributos: a observação do comportamento do sistema em tempo de execução; e a autonomia do *framework* na tomada de decisões a respeito do procedimento de *checkpoint*. O propósito da arquitetura é passar a responsabilidade das políticas de *checkpoint* – anteriormente dos desenvolvedores – para o próprio sistema. Assim, a DCA deverá ser capaz de verificar ocasiões favoráveis e desfavoráveis para *checkpoints*, submetendo-os quando necessário.

O uso de *checkpoint* dinâmico não é tratado como elemento de uso obrigatório, mas sim como uma opção ao usuário do Spark para uma tomada de decisões eficiente relacionada ao *checkpoint*. Sendo assim, a ativação do mecanismo segue o modelo dos atributos de configuração do *framework*, devendo-se indicar o atributo $spark.automatic.checkpoint$ no arquivo de configurações padrão. Além disso, a DCA não exclui a possibilidade da criação de *checkpoints* na forma tradicional, via código, já que seu comportamento não foi modificado.

O monitoramento da utilização de recursos é parte essencial do estabelecimento dinâmico de *checkpoints* no Spark. Para que a responsabilidade desse processo seja eliminada do desenvolvedor, foram criadas políticas de monitoramento capazes de identificar cenários em que o *checkpoint* deve ser considerado. Além disso, políticas de seleção de RDDs também foram definidas para escolher qual RDD deve ser salvo nos cenários de *checkpoint*.

As políticas possuem o objetivo de realizar escolhas automáticas e adaptadas ao contexto do sistema, relacionado tanto a sua utilização geral, quanto às características dos *datasets* usados. Para isso, são usados *internal agents* e *external agents* provenientes do módulo de monitoramento da arquitetura proposta. A implementação da DCA no Spark também conta com um elemento responsável pelo gerenciamento global das políticas: o *manager*.

3.5.2 Manager

O estudo dos fatores de observação por monitores externos tem uma noção global do Spark como uma ferramenta única, independente da quantidade de aplicações. Para que o monitoramento no *framework* ocorra da mesma maneira, e não individualmente para uma apli-

cação, é necessário o gerenciamento de alertas a partir do *driver*. Por isso, o gerenciador tem o papel de criar uma conexão com o *monitor* e tratar os alertas recebidos de *external agents*.

O *manager* também é responsável por implementar as políticas de seleção quando o usuário deseja visualizar o *framework* de forma independente das aplicações. Portanto, seu funcionamento inicia-se assim que o $ClusterManager$ é criado. Então, o *manager* espera por dois eventos: (a) o registro de aplicações, tão logo essas aplicações são iniciadas; (b) o recebimento de alertas do *coordinator* da DCA; e (c) recebimento de requisições das aplicações.

Com aplicações registradas, o *manager* pode realizar buscas por *datasets* na aplicação (através do contexto) ou diretamente nos *executors*, onde informações sobre blocos individuais são armazenadas. Assim que um alerta é recebido, uma busca é feita através das aplicações registradas para encontrar o *dataset* que se ajusta à política escolhida. Essa busca leva em conta todas as aplicações, mas define apenas um único *dataset* como alvo.

Ou seja, o *manager* acessa as aplicações e requisita o RDD mais apropriado para *checkpoint* de acordo com a política de seleção estabelecida. Ao final desse processo, o *driver* deve ter um RDD selecionado para cada aplicação. Novamente, de acordo com a política de seleção, os RDDs selecionados são filtrados para que apenas um seja escolhido. Esse procedimento garante que a política de seleção escolha o *dataset* mais apropriado dentre todos os RDDs disponíveis.

A partir da identificação do *dataset*, o *manager* é capaz de submeter seu *job* de *checkpoint* diretamente no $SparkContext$, fazendo com que essa submissão seja idêntica à tradicional definida em código. Além disso, uma aplicação independente pode também pode solicitar ao *manager* que a política de escolha definida seja feita com base em todas as aplicações registradas. Para isso, deve-se especificar o tipo de monitoramento das políticas de seleção através de um atributo nos arquivos de configuração do Spark. Uma política configurada com o atributo $local$ estabelece que a seleção fique restrita a RDDs da própria aplicação, sem passar pelo *manager*. Já o o atributo $global$ indica o uso do *manager*.

3.5.3 Políticas de Monitoramento

As políticas de monitoramento definem quando um procedimento de *checkpoint* deve ser iniciado no Spark, de modo a suprir alguma limitação observada. A principal vantagem da definição dessas métricas é a possibilidade de que momentos apropriados para salvamentos em *checkpoint* sejam escolhidos de forma automática. A escolha de qual política deve ser usada é definida em arquivos de configuração do Spark.

O momento da persistência de *datasets* em memória é fundamental para a eficiência de uma aplicação, em relação ao seu desempenho. Por isso, as políticas de monitoramento implementam um monitor dentro do próprio *framework*, com o intuito de identificar características de RDDs assim que eles são submetidos para armazenamento em memória e/ou processados. A função desse monitor é identificar situações em que o procedimento de *checkpoint* é favorável, de acordo com políticas estabelecidas. Para este caso, foram implementadas abordagens de: *threshold* de memória (*Memory Threshold*, ou MT), *threshold* de *lineage* (*Lineage Threshold*, ou LT) e custo computacional aliado à probabilidade de falhas (*Failure Awareness*, ou FA).

3.5.3.1 Memory Threshold

A utilização de dados em memória do Spark é um dos grandes benefícios de sua arquitetura, possibilitando um acesso mais eficiente aos dados. Porém, um bom gerenciamento de grandes *datasets* em memória é essencial para o desempenho de aplicações. Além de ser uma opção de tolerância a falhas, o procedimento de *checkpoint* é uma alternativa ao armazenamento em disco disponível para RDDs e pode beneficiar o desempenho de aplicações, principalmente em casos de recuperação pós-falha e quando a quantidade de memória é limitada.

A sobrecarga de dados em memória pode afetar negativamente o comportamento do Spark, dependendo da forma com que o desenvolvedor define o tratamento de *datasets* excedentes. Caso o método de persistência seja do tipo *memory-only*, por exemplo, o excedente é reprocessado sempre que é usado. Nesse sentido, a política de *threshold* de memória (*Memory Threshold*, ou MT) foi desenvolvida para evitar que a memória chegue a um nível de utilização alto. Nesse cenário, a partir do estabelecimento de um limiar máximo de consumo (tr, ou *threshold*), cenários de sobrecarga são identificados para auxiliar a decisão por *checkpoints*.

A política MT trabalha com duas possibilidades que devem ser indicadas nos arquivos de configuração do Spark: $global$ ou $local$. No caso do atributo $global$, existe uma visualização geral de *datasets* através do *driver*, tornando a seleção independente da aplicação. Neste caso, o *manager* é utilizado para pesquisar a situação de todas as aplicações registradas. Quando política de monitoramento indica a necessidade de um *checkpoint*, a política de seleção deve considerar todas as aplicações na escolha do *dataset*.

Já na opção local, a política MT age em dois momentos da execução de uma aplicação individual: quando um RDD é marcado para persistência através do método $persist()$ (ou $cache()$); e quando um *job* é finalizado. Em ambos os casos, o funcionamento da polí-

tica é o mesmo: verifica-se a quantidade de memória disponível, através do Spark Metrics, e condiciona-se o salvamento de um *checkpoint* ao *threshold* de memória estabelecido. Por isso, a partir do estabelecimento do *threshold*, cria-se a variante MT_{tr}, sendo tr o limiar escolhido.

Assim que a condição de *checkpoint* é verificada, a política determina qual *dataset* deve ser salvo através das políticas de seleção. Para isso, é necessário estabelecer a política de seleção nos arquivos de configuração do Spark. Caso não haja uma definição, definiu-se a política LRU – já usada pelo Spark – como a política de seleção padrão.

3.5.3.2 Lineage Threshold

Uma dos principais fatores que contribuem para alterações de desempenho em aplicações no Spark é a construção do *job* no que se refere às dependências entre RDDs. Quanto maior a complexidade de execução de um *job*, maior é o tempo de execução caso o *dataset* seja reutilizado. Em operações de recuperação pós-falha, esse fator também se mostra importante.

Deste modo, a política de monitoramento *Lineage Threshold* (LT) busca amenizar a complexidade de *jobs* com RDDs cujo *lineage* seja maior que um limite preestabelecido. A partir do salvamento desses *datasets* em *checkpoint*, seus *lineages* são truncados e perdem informações de dependência. Esse cenário pode se mostar atrativo quando o custo de leitura ou escrita de um RDD salvo em *checkpoint* é potencialmente menor do que sua recomputação.

Nessa política, o monitoramento não é periódico e só é iniciado no momento em que um RDD executa o método $rdd.persist()$ (ou o $rdd.cache()$). Esse monitoramento foi implementado no $SparkContext$, que possui uma estrutura de controle de RDDs persistidos. Após a execução do procedimento de persistência, a submissão de *checkpoint* do *dataset* é feita se o tamanho do seu *lineage* for maior que um *threshold* definido nos arquivos de configuração. Caso um cenário de *checkpoint* seja identificado, o RDD recebe o comando $rdd.checkpoint()$ logo após ser persistido.

O tamanho do *lineage* é definido pelo método $rdd.lineageSize$, que percorre a sequência de dependências presente na estrutura de dados de um RDD para observar a quantidade de *datasets* dependentes. Como cada RDD armazena apenas dependências diretas, a observação ocorre de forma recursiva: cada dependência é acessada e tem seu *lineage* calculado, até que o RDD raiz seja alcançado.

Dessa forma, tem-se uma visão completa do tamanho do caminho que um RDD percorre desde a sua fonte. Em caso de recuperação de partições desde a raiz, esse tamanho determina o

número mínimo de operações distintas que o Spark deve realizar para manter a disponibilidade dos dados. A eliminação de operações necessárias para reconstrução de um *dataset* é a principal justificativa de uso da política LT.

3.5.3.3 Failure Awareness

A técnica de *checkpoint* implementada no Spark possui cenários favoráveis de utilização. Um dos principais contextos é a presença de falhas no sistema. Com dados persistidos de forma local em cada *worker* – em memória e/ou em disco –, a tendência é de reprocessamento de dados perdidos à medida que eventos de falha permanentes são observados nos nodos. Desta forma, a política *Failure Awareness* (FA) é responsável por monitorar e prever possíveis cenários de falha durante a execução de um *job* no Spark. O objetivo é induzir o processo de *checkpoint* de um RDD quando este é propenso a ter dados perdidos e o custo de recomputação é maior que o custo de leitura do *checkpoint*.

Seu funcionamento consiste em uma verificação em duas etapas antes do RDD ser marcado para *checkpoint*. A primeira etapa diz respeito à probabilidade de falhas. O cálculo realizado verifica – a partir do histórico implementado pelo *coordinator* – os dados referentes às falhas no sistema, sendo eles o tempo médio entre falhas (MTBF) e o tempo decorrido desde a última falha observada (TSLF). Com estes dados, tem-se uma previsão do momento em que uma nova falha deve acontecer baseado em experiências anteriores.

Também utilizando o histórico, a política recupera o tempo estimado de conclusão do *job* baseado no *lineage* do RDD. O tempo é estimado de acordo com execuções anteriores do mesmo RDD. Dois RDDs são considerados idênticos quando a montagem da linha do tempo de um dos *datasets*, incluindo-se todas as dependências, é igual a do outro. Neste caso, sempre que um RDD é processado por uma ação, seu desempenho é registrado pelo histórico, conforme descrito na Seção 3.3.1.2.

Assim, em novas execuções, pode-se prever o tempo de execução de uma ação sobre determinado RDD (TE_{rdd}) considerando o número de partições do *dataset* a ser processado ($rdd.numPartitions$) e seu desempenho ($jobTime$, medido em tempo de execução por partição). O tempo TE_{rdd} de um *dataset* rdd é calculado pela Equação 3.5.

$$TE_{rdd} = jobTime * rdd.numPartitions \tag{3.5}$$

Com o tempo estimado de conclusão de um *job* e os fatores MTBF e TSLF, a primeira etapa da política é verificada. O andamento para a segunda etapa só é considerado caso a condição da Equação 3.6 seja satisfeita. Ou seja, quando TE_{rdd} for maior que o tempo esperado da nova falha (que consiste no MTBF, retirando o TSLF), entende-se que o *job* poderá sofrer uma falha durante um processamento futuro. Neste caso, ter um *dataset* em *checkpoint* pode ser determinante para uma rápida recuperação de partições perdidas em caso de reuso.

$$TE_{rdd} > MTBF - TSLF \tag{3.6}$$

Após a validação da primeira condição, há uma nova condição para garantir que um RDD não seja submetido a *checkpoint* quando seu custo computacional é menor que seu custo de leitura. Essa condição é verificada para evitar que RDDs de pequena complexidade sejam submetidos a *checkpoint*, quando esse procedimento tende a degradar o desempenho da aplicação sem retornar uma eficiência de recuperação que justifique o uso do *checkpoint*. Ou seja, quando uma falha acontece e a recomputação das partições perdidas for mais rápida – baseado no TE_{rdd} estimado –, o *checkpoint* é evitado.

Para isso, um terceiro fator de observação é considerado: o tempo médio de leitura no HDFS, obtido a partir da API de métricas disponibilizada pelo HDFS (conforme seção 2.2.4). O acesso à API é feito por uma requisição GET a métricas de *DataNodes*, a fim de obter o atributo $ReadBlockAvgTime$ do HDFS. O dado retornado informa o tempo médio de leitura de um bloco no HDFS. Uma vez que o número de partições transforma-se no número de blocos do *dataset* quando salvo em *checkpoint*, pode-se estabelecer o tempo de leitura do RDD no HDFS (HRT_{rdd}) através da Equação 3.7.

$$HRT_{rdd} = ReadBlockAvgTime * rdd.numPartitions \tag{3.7}$$

Logo, o *checkpoint* é submetido caso a condição da Equação 3.8 seja satisfeita. Isto é, uma marcação de *checkpoint* é realizada quando o tempo estimado de execução de um *dataset* (TE_{rdd}) é maior que o tempo de leitura do mesmo no HDFS (HRT_{rdd}). Caso contrário o salvamento é evitado, pois entende-se que o *checkpoint* pode não trazer um retorno satisfatório dado o contexto do sistema.

$$TE_{rdd} > HRT_{rdd} \tag{3.8}$$

3.5.4 Políticas de Seleção

As políticas de seleção são métricas auxiliares, pois dependem das políticas de monitoramento para serem executadas. Dessa forma, assim que um dos monitores identifica um cenário de *checkpoint*, deve-se tomar uma medida para definir qual dos *datasets* deve ser salvo. Nesse cenário, foram estabelecidas as políticas de seleção $Individual$, LRU e LL.

3.5.4.1 Individual

Quando as políticas de monitoramento são enfatizadas em um *dataset* específico, a seleção do RDD já é definida. Esse é o caso das políticas de monitoramento LT e FA, além da política de *checkpoint* estática. Assim, a política de seleção $Individual$ consiste em decidir por submeter um *job* de *checkpoint* para o RDD já escolhido por políticas de monitoramento – ou pelo desenvolvedor, no caso do *checkpoint* estático. A Figura 3.3 exibe o fluxo de etapas desde sua requisição até o *job* de *checkpoint*.

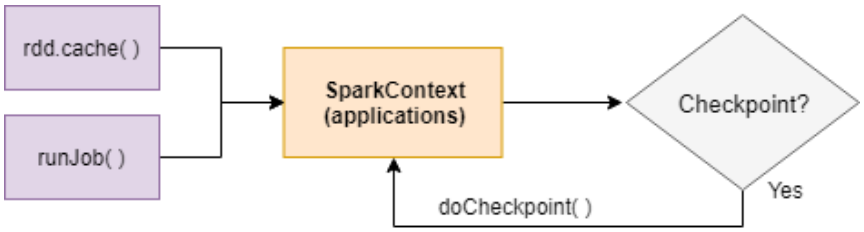


Figura 3.3 – Fluxo de etapas do *checkpoint* automático com a política de seleção $Individual$.

O fluxo é simples, já que consiste apenas em sua revisão pelo $SparkContext$ para definir se o *dataset* deve ser salvo em *checkpoint*. Nesse caso, nenhuma comunicação com o *manager* é necessária. Assim, a observação é feita pela arquitetura DCA sempre que um RDD é marcado para armazenamento em memória e, também, imediatamente depois que um *job* é processado através do método $runJob$. No caso de política estática, a seleção ocorre para os *datasets* marcados manualmente para *checkpoint* via código. Desta forma, se o *dataset* deve ser salvo, o contexto chama o método $doCheckpoint()$ e encerra o fluxo de etapas.

3.5.4.2 LRU

A política de seleção LRU (*Least Recently Used*) é uma abordagem originalmente utilizada pelo Spark para transferir blocos de *datasets* da memória para o disco, quando o nível de persistência usada permite. A abordagem consiste em selecionar o *dataset* menos utilizado

através de uma questão temporal. Isto é, a decisão pelo menos recentemente utilizado escolhe o *dataset* que tem o maior tempo decorrido desde seu último acesso (de leitura ou escrita).

A implementação do LRU no Spark é baseada na estrutura de dados $LinkedHashMap$ do Java, que define um mapeamento de *hash* que pode ter o atributo $accessOrder$ ativado para criar-se uma ordenação temporal. A cada novo acesso em um elemento, este passa a ocupar a primeira posição da estrutura, mantendo a ordem sempre definida. Essa estrutura é usada para armazenar os blocos de *datasets* pelo Spark, podendo-se encontrar o último bloco acessado ao verificar a primeira posição.

A política LRU foi modificada para auxiliar a arquitetura DCA. Conforme demonstrado pela Figura 3.4, seu procedimento começa pelo $SparkContext$ com uma requisição síncrona para cada Backend registrado. No $Backend$, uma nova requisição síncrona é feita para cada *executor* registrado. O *executor* acessa seu $BlockManager$ e envia como resposta o id do RDD correspondente e o tempo do seu último acesso.

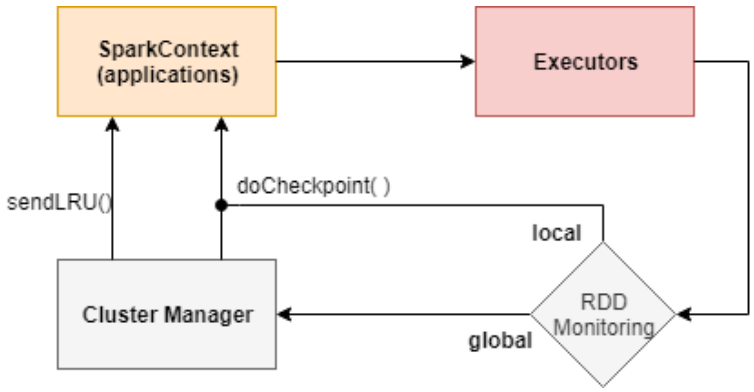


Figura 3.4 – Fluxo de etapas do *checkpoint* automático com a política LRU.

A partir da resposta de todos os *executors*, o Backend retorna o RDD menos recentemente utilizado ao $SparkContext$. O *manager*, quando o monitoramento de RDDs $global$ é usado, realiza o mesmo procedimento de escolha para cada mensagem de retorno dos *executors* usando o tempo de acesso como parâmetro. Neste caso, o RDD definido como LRU global é identificado e uma requisição do *manager* ao Backend do *dataset* é feita com a mensagem $doCheckpoint()$. Sem o *manager*, a mensagem é enviada diretamente ao *context*.

3.5.4.3 Longest Lineage

A técnica de seleção LL (*Longest Lineage*) toma como parâmetro o tamanho do *lineage* de um RDD. A abordagem é classificatória e exclusiva, selecionando apenas o *dataset* com

o maior número de dependências entre todos. Seu uso pode ser importante para a decisão de seleção quando apenas um único RDD deve ser escolhido e o usuário preza por eliminar conjuntos de dados complexos. Ainda que seja uma abordagem de escolha simples, selecionar o RDD com maior *lineage* pode ser considerada uma opção com potencial para salvar *datasets* complexos em detrimento de RDDs com poucas dependências.

A Figura 3.5 mostra o fluxo de operações da política LL. A sequência de operações de busca para esta métrica se dá diretamente no $SparkContext$ correspondente, que percorre sua lista de RDDs persistidos e acessa o método $rdd.lineageSize$. O *dataset* com o maior número dependências é escolhido.

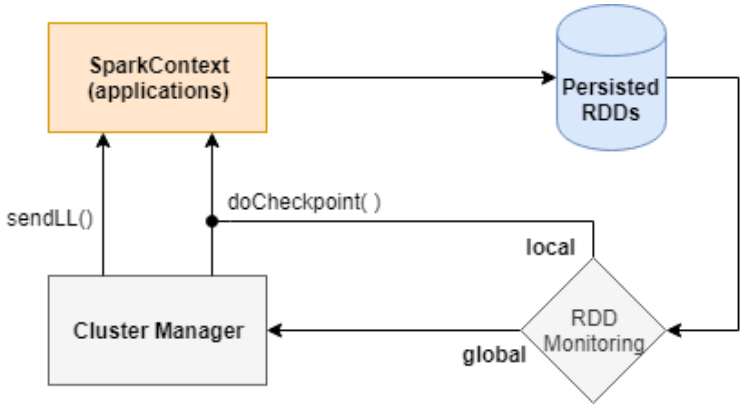


Figura 3.5 – Fluxo de etapas do *checkpoint* automático com a política de seleção LL.

No caso de monitoramento $global$ pelo *supervisor*, o $SparkContext$ envia o id do RDD como resultado ao *manager*, bem como o tamanho do *lineage*. A escolha do *manager* leva em consideração o RDD com maior *lineage* dentre os recebidos pelas aplicações. A partir das informações recebidas, o *master* acessa o Backend responsável pelo *dataset* escolhido e requisita o *checkpoint* do RDD com a mensagem $doCheckpoint()$. Em caso de monitoramento de RDDs $local$, o próprio *context* chama o método $doCheckpoint()$, já que o acesso ao *manager* é dispensável.

4 VALIDAÇÃO

A fim de validar a proposta da DCA e as implementações desenvolvidas, este capítulo apresenta os testes de desempenho feitos em aplicações com os *frameworks* Apache Hadoop e Apache Spark. As experimentações foram realizadas na plataforma Grid'5000[2]: uma plataforma para experimentos, apoiada por um grupo de interesses científicos, hospedada por Inria, CNRS, RENATER e diversas Universidades, bem como outras organizações. O *grid* oferece um ambiente distribuído de larga escala com infraestrutura para testes relacionados a aplicações distribuídas e paralelas (BALOUEK et al., 2013).

A infraestrutura usada pelo Grid'5000 é dividida em diferentes sites, que hospedam *clusters* de máquinas com uma grande diversidade de configurações de *hardware*. Para as validações, foram usadas as máquinas de três diferentes *clusters*. O *cluster* $paravance$ conta com 2 CPUs Intel Xeon E5-2630 v3 (8-core), 128GB de RAM e 558GB disponível em disco. O *cluster* $grisou$ conta com as mesmas especificações, de modo que esses dois foram usados para validações com o Spark. Já o *cluster* $graphene$, usado para validar o Hadoop, possui um processador Intel Xeon X3440 (4-core), com 16GB de memória RAM e 298GB de disco. Foram utilizadas 8 máquinas de cada cluster para o ambiente de testes de ambas as validações.

Cenários de falhas (CF) foram definidos para criar a condição de falhas em elementos dos *frameworks* usados. Assim, o objetivo foi analisar o comportamento das etapas de recuperação frente a eventos de falha. No Hadoop, CFs foram definidos para emular situações em que o elemento mestre do HDFS é induzido a uma falha transiente, onde seu processo é encerrado em tempo de execução através do comando $kill$ do Linux e, logo em seguida, é reiniciado manualmente – conforme descrito na seção 3.4.3. A indução caracterizada como cenário de falha transiente permite que a recuperação do HDFS seja avaliada a partir do tempo de execução excedente, adicionado pela carga de leitura do *namespace* salvo pelo *checkpoint* mais recente.

Já no Spark, os CFs foram criados para emular falhas nos elementos trabalhadores do *framework*: os *workers*. As falhas são permanentes, de modo que um elemento falho não volta a ser iniciado novamente. Para isso, o comando $kill$ também é usado no processo de um ou mais *workers*. Esse tipo de falha demanda procedimentos de reprocessamento de dados pelo Spark na etapa de recuperação, podendo-se avaliar o desempenho das diferentes políticas a partir do tempo excedente de processamento.

[2] Mais detalhes em https://www.grid5000.fr

4.1 Benchmarks

Para uma avaliação completa da DCA no Hadoop e no Spark, foram criados diferentes cenários de teste através de três *benchmarks* com diferentes propósitos: TestDFSIO, PageRank e K-Means. O TestDFSIO é uma aplicação *I/O intensive* que realiza testes de leitura e escrita no HDFS. Esse *benchmark* foi validado no Hadoop, como forma de avaliar o desempenho dos salvamentos de *checkpoint* estaticamente e dinamicamente configurados.

A validação do Spark contou com os *benchmarks* PageRank e K-Means. Ambos são considerados *CPU intensive* com características iterativas. Apesar das semelhanças, as implementações dos *benchmarks* no Spark diferem em termos de código. No PageRank, poucos agrupamentos são feitos entre iterações, diferentemente do K-Means que possui agrupamentos em maior quantidade e complexidade. Os dois *benchmarks* foram executados com o auxílio da plataforma HiBench (HUANG et al., 2010), que oferece facilidades para a execução e para testes de desempenho em aplicações distribuídas.

4.1.1 TestDFSIO

O TestDFSIO é um *benchmark* para análise do desempenho do HDFS em relação à entrada e saída de arquivos. Esse tipo de teste é amplamente usado para criar condições de estresse de *I/O* no HDFS. Assim, pode-se fazer uma avaliação do sistema de arquivos em situações de sobrecarga de leitura e de escrita de dados no HDFS. O *benchmark* é caracterizado como uma aplicação *I/O Intensive*.

Por isso, o TestDFSIO pode ser executado para dois propósitos diferentes: escrita e leitura. O processo de escrita é realizado com um mapeamento 1-1 de *splits* e arquivos. Isto é, a quantidade de arquivos gerados determina o número de divisões dentro do HDFS. Consequentemente, esse número determina a quantidade de tarefas de mapeamento realizadas pelo *benchmark*. No HDFS, os arquivos recebem o mesmo nome mas são diferenciados pela adição do identificador da divisão. O conteúdo dos arquivos é gerado de forma randômica até que o tamanho definido seja alcançado.

Na execução de leitura, o *benchmark* acessa um determinado conjunto de dados já armazenado no HDFS e imprime o resultado na tela. Os dados não são gerados durante esse procedimento, de modo que o *executor* deve gerar os dados previamente. A geração de dados pode ser feita manualmente ou através do próprio *benchmark* na operação de escrita. Assim

como na escrita, o número de arquivos a ser manipulado determina a quantidade de tarefas de mapeamento.

4.1.2 Pagerank

O PageRank é um algoritmo iterativo de classificação de páginas Web, proposto pela Google para facilitar e otimizar seu processo de busca (ALTMAN; TENNENHOLTZ, 2005) (PAGE et al., 1999). A classificação ocorre por meio de diversas comparações entre páginas com conteúdo semelhante a fim de determinar níveis de importância. Os níveis são quantificados e estabelecidos dentro do intervalo que vai de 0 (nada relevante) a 10 (extremamente relevante). Assim, o usuário tem uma hierarquia de relevância definida ao realizar uma busca por determinado conteúdo, de modo que o agente buscador pode optar por ordenar o resultado de acordo com seu nível.

A relevância é calculada com base nos apontamentos (isto é, *links*) que uma página recebe de outras páginas. Um apontamento de uma página para outra é considerado um "voto", de modo que o processo pode ser entendido como uma votação entre documentos semelhantes. Assim, quanto maior for o número de referências a uma página, maior deve ser sua relevância dentro do próprio escopo. Além da quantidade, o PageRank também considera a qualidade das referências que uma página recebe. Ou seja, quanto maior é a relevância da página que está indicando, mais relevante é o seu voto e mais valiosa é a pontuação da página indicada.

A iteratividade característica do algoritmo do PageRank serve para rebuscar a confiabilidade das votações feitas. Em cada iteração, um novo cálculo de relevância é feito tendo como base o cálculo anterior, de modo que todas as páginas iniciam a primeira iteração com a mesma relevância: $1/n$, sendo n o número total de páginas. Como o algoritmo considera a relevância de uma página para definir o peso de seus votos, mais iterações possibilitam a qualificação do processo de votação.

4.1.3 K-Means

O K-Means é um algoritmo de agrupamento iterativo que consiste no particionamento de valores em k *clusters* baseado na distância entre os pontos. Procedimentos de agrupamento são frequentemente utilizados pela necessidade de classificação de dados para a obtenção de informações (GOPALANI; ARORA, 2015). No cenário atual, em que uma grande quantidade de dados é gerada, o K-Means se porta como um dos algoritmos de classificação/agrupamento

mais bem utilizados por não depender de pré-classificações. Por isso, esse algoritmo é definido como não supervisionado.

O agrupamento é feito para que k *clusters* sejam formados, de modo que o número de *clusters* deve ser informado pelo usuário. Cada *cluster* é definido por seu centróide: um valor central presente no espaço de valores usados, mas que não representa (necessariamente) um ponto do arquivo de entrada. De forma inicial, os k centróides são definidos aleatoriamente, sendo que o HiBench emprega uma função de distribuição normal para definir a posição dos centróides dentro do espaço usado.

Além disso, o HiBench realiza o cálculo de centróides a partir de i etapas de inicialização baseado no algoritmo K-Means++ (ARTHUR; VASSILVITSKII, 2007). Em cada etapa, o cálculo dos pontos é otimizado para que se encontre pontos centrais eficientes. O uso do K-Means++ na estimativa dos centróides foi proposto para otimizar o resultado final do K-Means, de modo a corrigir os problemas de desempenho e qualidade de saída do K-Means tradicional (BAHMANI et al., 2012). Por padrão, 2 etapas de inicialização são executadas para a criação dos centróides no HiBench.

Em cada iteração, o algoritmo calcula a distância que os pontos de entrada possuem dos centróides. Um ponto passa a ser classificado do *cluster* com centróide menos distante. Após a classificação de todos os pontos em uma iteração, o algoritmo realiza a média de valores dos pontos de cada *cluster* e o valor da média calculada é estabelecido como o novo centróide. Para finalizar a iteração, o K-Means verifica a soma do quadrado da distância que cada centróide variou. Uma nova iteração é iniciada quando essa soma é maior que um *threshold* pré-determinado para todos os centróides. Um número máximo de iterações também pode ser definido para o controle de parada do algoritmo.

4.2 Validação no Hadoop

Esta seção apresenta os resultados obtidos pela validação do mecanismo de configuração dinâmica para *checkpoints* no *framework* Apache Hadoop. O *benchmark* TestDFSIO foi usado para avaliar o HDFS – elemento chave do *checkpoint no Hadoop* – em sitações de estresse de *I/O*. Os testes foram divididos de acordo com o mecanismo de configuração usado. No mecanismo estático, o comportamento padrão do Hadoop é usado com definições manuais e estáticas do período entre *checkpoints*. Já o mecanismo dinâmico foi usado para configurar – em tempo real – novos períodos de *checkpoint* de acordo com métricas de monitoramento.

Os CFs foram definidos com três cenários baseados no número de falhas por execução: 1 (CF_1), 2 (CF_2) e 3 (CF_3) falhas. O momento de indução das falhas é escolhido de forma pseudo-randômica por rodada de execução, com intervalos baseados nos *baselines* – execuções com configuração estática padrão: 3600 segundos – sem falhas de cada cenário de teste. Os *baselines* servem como sustentação para a análise de desempenho das variações usadas. A primeira falha ocorre entre 25% e 50%, a segunda entre 50% e 60% e a terceira entre 60% e 75% do tempo a partir do início do *baseline*. No mecanismo dinâmico, o momento de indução também é relativo ao *baseline* da configuração estática padrão (3600 segundos), já que a arquitetura proposta não é executada em um cenário sem falha.

Os testes foram divididos em três etapas: inicialmente, o Hadoop com configurações estáticas (mecanismo padrão) em um cenário sem nenhum tipo de falha induzida. Além disso, os testes com configuração estática também descrevem execuções sobre os 3 cenários de falha descritos. Em relação às configurações de *checkpoint*, foram definidos 4 cenários estáticos usando os seguintes períodos: 3600, 360, 36 e 10 segundos. Em seguida, os testes do mecanismo de configuração dinâmica são apresentados sobre os três cenários de falha.

Como forma de definir novos períodos entre *checkpoints*, o *monitor* da DCA foi usado com as aproximação de Young (YOUNG, 1974) e de Daly (DALY, 2006). As aproximações realizam o cálculo de períodos ótimos de *checkpoint* baseados no custo de *checkpoints* e na análise de confiabilidade – a partir no tempo médio entre falhas – do sistema. Uma vez que as métricas usadas no mecanismo dinâmico necessitam da análise desses fatores, cenários sem falhas não foram validados para este caso.

Os testes propostos utilizaram a aplicação TestDFSIO para a avaliação dos mecanismos dinâmico e estático com grandes cargas de operação em disco. O *benchmark* foi usado para a escrita de arquivos com dimensão de 8GB, variando-se a quantidade de arquivos gerados por execução entre 24, 32 e 40 arquivos. Logo, o tamanho total envolvido foi de 192GB, 256GB e 320GB, respectivamente.

4.2.1 Cenário Estático

Para avaliar o mecanismo de configuração estática de *checkpoints* no Hadoop, foram realizados testes de desempenho dos *benchmarks* através de diferentes configurações iniciais de período entre *checkpoints*. Os testes iniciais analisam o desempenho do mecanismo estático em um cenário sem falhas para determinar o nível de intrusividade alcançado pelos procedimentos

de *checkpoint* ao longo da execução dos *benchmarks*.

Assim, pode-se ter uma noção da sobrecarga gerada através da variação de frequência de salvamento dos *checkpoints*. Os cenários de falha também foram executados, com o propósito de verificar o desempenho de recuperação do HDFS quando seu elemento mestre falha. Nesse sentido, a técnica de *checkpoint* é acionada para realizar este procedimento e a variação do período entre *checkpoints* é analisada a seguir.

Os resultados obtidos em relação à execução do *benchmark* são sumarizados na Tabela 4.1, que exibe o tempo de execução, em segundos, para cada configuração de número de arquivos, cenário de falha e período estático usado. Além disso, o número de *checkpoints* realizados (NCp) é descrito de acordo com a média de salvamentos por rodada.

Cenário de Falha	**Período (s)**	**Carga de dados**					
		20x8GB		**32x8GB**		**40x8GB**	
		Execução (s)	NCp	**Execução (s)**	NCp	**Execução (s)**	NCp
S/falha	3600s	1542,2	0,2	1846,8	0,6	2148,7	0,6
	360s	1580,3	4,2	1998,0	5,4	2281,7	5,3
	36s	1611,5	35,6	2056,2	54,1	2340,5	58,2
	10s	1625,4	94,7	2126,8	178,7	2386,7	198,5
CF_1	3600s	1715,0	0,5	2236,1	0,4	2419,0	0,33
	360s	1677,2	4,7	2061,1	4,7	2389,8	4,3
	36s	1649,4	40,0	2098,5	42,8	2480,9	52,12
	10s	1651,0	118,0	2152,2	175,3	2413,1	186,6
CF_2	3600s	1743,2	0,6	2241,1	0,3	2490,7	0,3
	360s	1681,4	3,5	2154,2	3,7	2394,6	3,8
	36s	1668,9	37,56	2115,6	39,8	2419,2	56,79
	10s	1675,5	105,33	2200,3	141,6	2428,3	181,6
CF_3	3600s	1807,5	0,5	2272,5	0,3	2530,3	0,4
	360s	1720,1	3,5	2168,6	3,5	2482,1	4,1
	36s	1677,0	31,0	2157,6	33,8	2477,6	41,8
	10s	1694,5	88,6	2201,1	109,6	2480,5	179,6

Tabela 4.1 – Desempenho do mecanismo de configuração estática para *checkpoints* nos cenários de teste do TestDFSIO.

É possível notar pela Tabela 4.1 que o aumento da frequência de *checkpoints* estáticos é impactante no desempenho da aplicação, uma vez que também aumenta o número de operações de *I/O* realizadas pelo HDFS. Dado que as operações de *checkpoint* incluem transferência de dados entre NN e SNN, além do processamento do novo `FSImage`, nota-se que *checkpoints* mais frequentes são mais custosos devido à sobrecarga gerada pela comunicação entre elementos. Já os *checkpoints* menos frequentes, mesmo que necessitem salvar mais transações por *checkpoint*, não ocasionam sobrecarga de comunicação.

Também foi definida uma medida de sobrecarga (*overhead*), referente ao acréscimo de tempo observado por determinado caso, com base em outro. O comportamento da sobrecarga, conforme variou-se o cenário de falhas, mostrou duas características distintas. Em princípio, a indução de uma única falha (CF_1) aumentou consideravelmente o tempo de execução do *benchmark* em todos os cenários de carga. Porém, na segunda e na terceira falhas (cenários CF_2 e CF_3), a sobrecarga não se mostrou tão relevante. Esse reflexo é observado pois a recuperação do $NameNode$ exige um salvamento de *checkpoint* no momento de reinício do elemento falho. Ou seja, falhas próximas entre si tendem a perder pouco trabalho.

Ainda que a configuração com *checkpoints* mais frequentes possua uma perspectiva de recuperações eficiente, o tempo de execução final em cenários de falha é maior que configurações com 360 segundos e 36 segundos quando apenas uma falha é induzida por execução. Este é um comportamento previsto devido à alta sobrecarga de salvamentos em momentos livres de falha, o que acaba prejudicando o andamento da aplicação independentemente da eficiência de recuperação. Por isso, em cenários de alta confiabilidade do sistema, a alternativa mais viável para a escolha de um período estático deve seguir uma abordagem pouco intrusiva de *checkpoints*.

Por outro lado, aumentando-se o número de falhas simultâneas por execução, configurações com recuperações eficientes tendem a produzir resultados mais satisfatórios. Isto é, a demanda por procedimentos de recuperação é maior com uma quantidade maior de falhas, por isso o estado de atualização do `FSImage` pelo *checkpoint* torna-se essencial nesse caso. Assim, entende-se que alternativas mais cautelosas – com *checkpoints* realizados com maior frequência – são a melhor solução para uma definição estática do período em caso de baixa confiabilidade do sistema.

4.2.2 Cenário dinâmico

O mecanismo configuração dinâmica foi usado com as aproximações de Young (YOUNG, 1974) e de Daly (DALY, 2006) para o cálculo de períodos ótimos de *checkpoint* baseados no custo de *checkpoints* e na análise de confiabilidade do sistema – com base no tempo médio entre falhas. Com o uso da DCA, novos períodos são definidos através das comunicações entre HDFS, *coordinator* e *monitor*.

Além disso, o histórico de atributos do *coordinator* foi usado para determinar, em tempo-real, o comportamento dos fatores MTBF e custo de *checkpoint*. Contudo, o cenário sem falhas

não foi executado para o mecanismo dinâmico pela falta de informação *a-priori* sobre a quantidade de falhas, o que impossibilita o uso das fórmulas de Young e Daly.

Inicialmente, o mecanismo de configuração dinâmica foi executado usando-se uma abordagem inicial para o histórico. Nessa abordagem, o conteúdo do histórico inicia-se vazio, de modo que as informações são obtidas no decorrer das execuções. Os N cenários de falha CF_N iniciam com o período padrão de *checkpoint*: 3600 segundos. Isto é, a base de dados usada para o histórico é vazia: DS_{Empty}. Ao final, cada cenário de falha tem uma base de dados consolidada por E execuções. Assim, cria-se as bases de dados DS_N para cada cenário de falha CF_N.

A abordagem inicial também contou com uma definição padrão para a janela de observação: 20 entradas no histórico. Os resultados, exibidos na Tabela 2, são divididos pela aproximação usada (App) e indicam o tempo de execução para cada cenário de falhas e o número de *checkpoints* realizados (NCp).

Cenário de Falha	App	**Carga de dados**					
		20x8GB		**32x8GB**		**40x8GB**	
		Execução (s)	NCp	**Execução (s)**	NCp	**Execução (s)**	NCp
CF_1	Young	1640,7	5,0	2066,8	7,5	2345,4	8,4
	Daly	1654,6	4,2	2064,9	6,7	2339,5	7,7
CF_2	Young	1670,6	5,4	2093,0	8,4	2363,1	9,5
	Daly	1679,7	7,5	2129,5	7,6	2384,1	6,9
CF_3	Young	1738,9	6,4	2138,2	9,4	2398,8	10,6
	Daly	1738,9	7,6	2151,3	8,5	2419,2	8,8

Tabela 4.2 – Desempenho da DCA com as aproximações de Young e Daly nos cenários de teste do TestDFSIO.

Os dois fatores usados nas métricas de Young e Daly são o custo de *checkpoint* (referente à intrusividade de salvamentos em momentos livres de falha) e o MTBF (relacionado à eficiência de recuperação). Por consequência, adaptações baseadas no histórico conseguem equilibrar com sucesso o período de acordo com comportamentos sempre atuais do sistema. Assim, a aplicação do mecanismo de configuração dinâmica auxilia em casos de indecisão ou desconhecimento do comportamento do sistema.

Situações de baixa confiabilidade, em que a ocorrência de falhas é frequente, podem ser identificadas quando as aproximações de Young e Daly são usadas, devido ao fator MTBF. Os cenários de intrusividade também são identificados e amenizados pelas aproximações, já que o custo do procedimento também é considerado. Ao contrário dos casos de configurações

estáticas, o número de *checkpoints* realizados é maior com mais falhas quando o período é dinamicamente configurado.

Para comparar os resultados obtidos pelas configurações estáticas e dinâmicas, a Figura 4.1 apresenta os tempos de execução obtidos. Nesse caso, o cenário de carga com 20 arquivos foi usado como parâmetro de comparação. É possível notar que a configuração dinâmica baseada nas aproximações de Young e Daly manteve-se com tempos de execução abaixo dos casos configurados de forma estática.

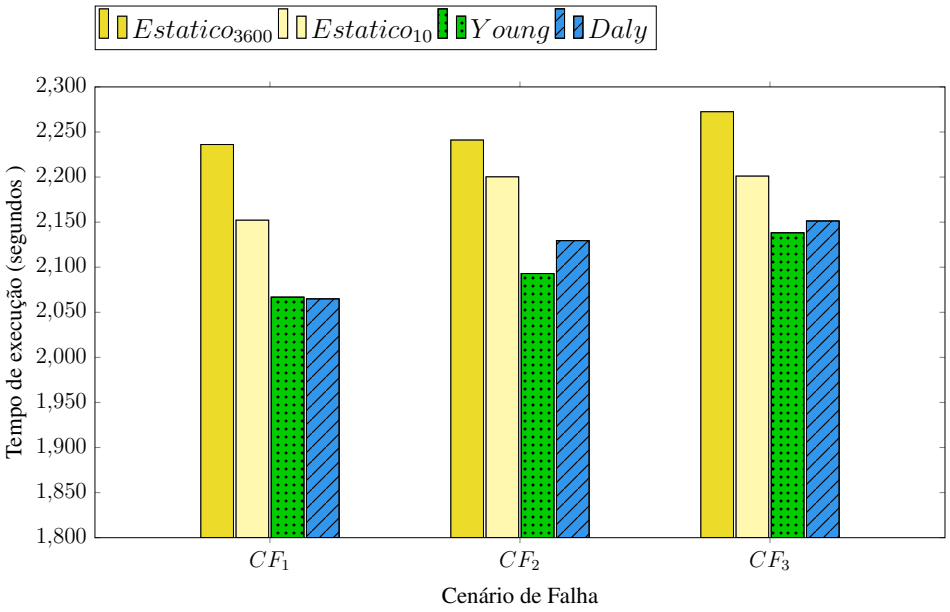


Figura 4.1 – Tempo de execução do $TestDFSIO$ com configurações estáticas e dinâmicas.

A grande vantagem da execução com o mecanismo proposto é a adaptação em tempo real ao comportamento do sistema, ajustando-se o período de *checkpoint* de acordo com embasamentos já validados. De fato, os estudos para a aplicação de períodos para o *checkpoint* alcançaram resultados próximos ao ideal – quando o tempo de recuperação tende a zero.

Os resultados também mostram que os salvamentos de *checkpoint* em cenários livres de falha não incluíram uma intrusividade de escrita em disco, como no caso do *checkpoint* estático em 10 segundos. Ainda, as etapas de recuperação se mostraram tão eficientes quanto o caso com *checkpoints* frequentes, o que indica uma alta confiabilidade atingida.

A eficiência dos casos com configuração dinâmica se dá pela quantidade adequada de *checkpoints* realizados por execução. A Tabela 4.3 mostra essa relação a partir da quantidade média de *checkpoints* salvos nas rodadas de teste (NrC). O número de vezes em que um novo período entre *checkpoints* foi enviado do *coordinator* para o HDFS também é apresentado, que também corresponde às mudanças de contexto observadas pelo monitoramento (CCh). Os

dados são divididos pelos cenários de falha usados.

Configuração	CF_1		CF_2		CF_3	
	NrC	CCh	NrC	CCh	NrC	CCh
$Estatico_{3600}$	0,6	0	0,6	0	0,7	0
$Estatico_{10}$	201,5	0	186,1	0	181,5	0
$Young$	4,0	7,8	5,1	6,1	5,1	7,1
$Daly$	5,1	9,3	4,8	5,8	4,2	5,9

Tabela 4.3 – Média de *checkpoints* salvos (NrC) e mudanças de contexto (CCh) por execução.

As abordagens de Young e Daly no mecanismo de configuração dinâmica habilitaram o Hadoop a executar um número menor de *checkpoints* em relação à configuração estática com salvamentos mais frequentes. Com ambas as aproximações, cerca de cinco procedimentos de *checkpoint* foram realizados em cada rodada de teste. Essa quantidade mostrou-se suficiente para garantir a eficiência de recuperação esperada, mesmo com uma frequência de salvamentos consideravelmente menor. Desta forma, nota-se que as abordagens de cálculo dinâmico obtiveram sucesso na busca pelo equilíbrio entre desempenho e confiabilidade almejado pela DCA.

A partir do número de *checkpoints* e dos tempos de execução obtidos, percebe-se que a posição de salvamentos é induzida a acontecer em momentos estratégicos das execuções. Uma vez que as aproximações de período ideal usadas consideram o tempo médio entre falhas e o custo de *checkpoints*, a previsão desses valores pelo histórico mostrou uma eficiência satisfatória. Ou seja, apesar da simplicidade das fórmulas, ter os fatores M e C como base para calcular períodos equilibrados de *checkpoints* tende a gerar resultados eficientes.

A Figura 4.2 exibe os períodos de *checkpoint* calculados e definidos pelo mecanismo de configuração dinâmica, separados em 10 etapas de execução. Cada etapa refere-se a 2 rodadas de teste, de modo que os períodos mostrados na Figura 4.2 correspondem aos períodos usados no início de cada etapa. Neste caso, apenas os cenários com histórico baseado em DB_{Empty} foram testados, por isso todas as execuções iniciam com a mesma configuração: 3600 segundos.

É possível verificar que não há um período ideal fixo encontrado pelas aproximações usadas. Porém, existe uma tendência de padronização do período entre os cenários de falha à medida que mais etapas são executadas. Durante as primeiras execuções, o histórico possui informações sem nenhum tipo padrão reconhecido. Conforme novas entradas são adicionadas, mais dados são armazenados. Consequentemente, a análise de custos torna-se mais aprimorada com mais observações feitas.

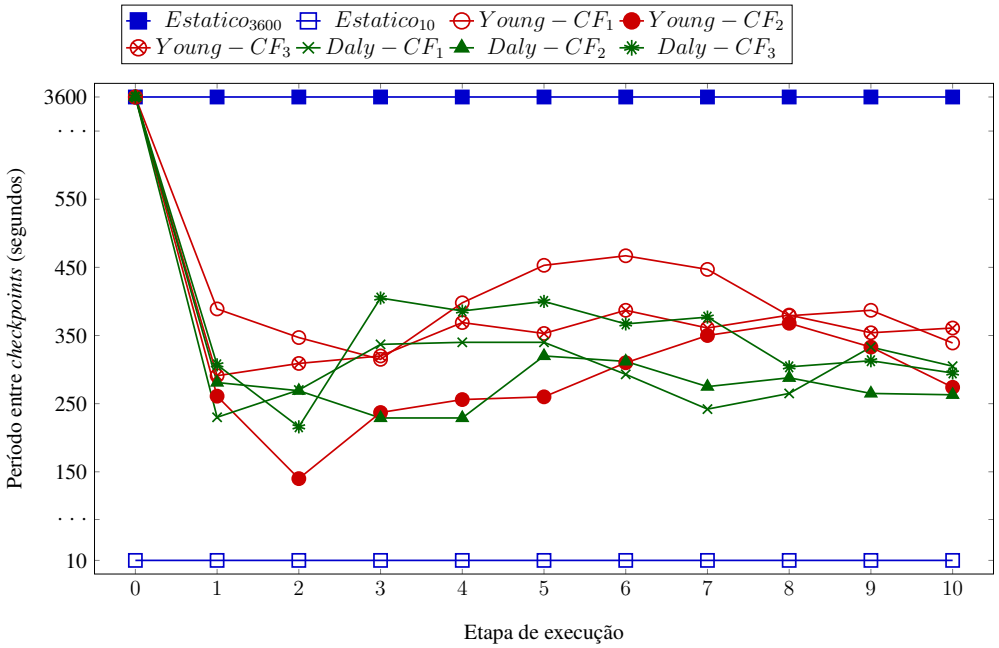


Figura 4.2 – Períodos de *checkpoint* escolhidos pelas aproximações durante os testes divididos em etapas de 2 execuções.

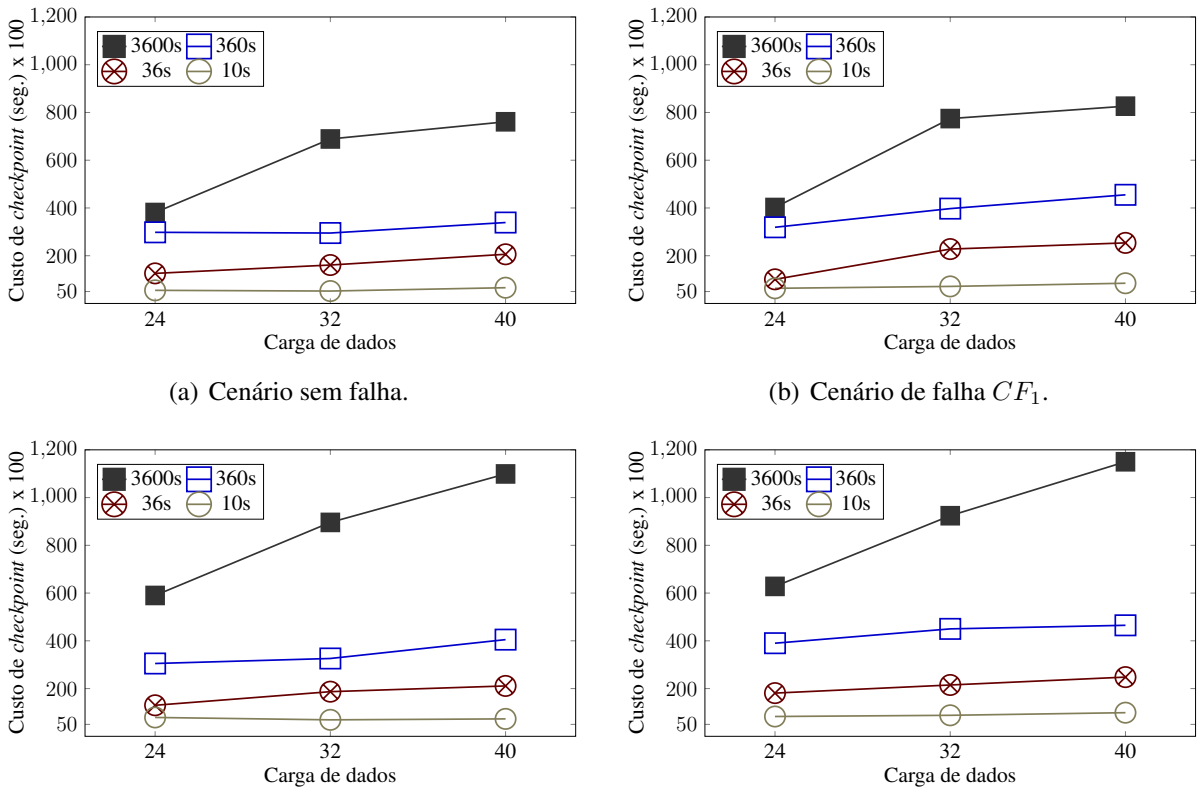
Ainda que não exista um padrão para a definição de um período fixo, há também uma tendência para escolhas de períodos entre 250 segundos e 400 segundos. Essa característica mostra como a definição estática de um período ideal de *checkpoint* é inviável. Isto é, o período ideal muda constantemente durante a execução do *framework* – a Tabela 4.3 mostra que 5 a 9 mudanças são esperadas por rodada do *benchmark* usado. Mudar este atributo nos arquivos de configuração sempre que um novo valor é escolhido impossibilita o funcionamento do *framework*, que teria seus serviços reiniciados muitas vezes.

No cenário FS3, em que o tempo de indução das falhas não é estabilizado, existe uma grande diferença entre os períodos escolhidos. Esse comportamento é devido aos diferentes tempos de indução da falha, que acabam indicando valores diferentes de MTBF e de período entre *checkpoints*. Consequentemente, os custos dos *checkpoints* também acabam gerando valores diferentes a cada execução. Ainda assim, a adaptação pelo histórico também é favorável a uma padronização do período conforme a base de dados é incrementada.

4.2.3 Custos

O custo de *checkpoint* é calculado pelo tempo de execução necessário para que o NN atualize a imagem de *checkpoint*, desde o envio do *log* de edições ao SNN até o recebimento da imagem atualizada. A Figura 4.3 apresenta uma análise referente ao custo estimado da rea-

lização do procedimento de *checkpoint* nos diferentes cenários de teste. Nesse caso, o histórico do mecanismo dinâmico foi usado mesmo nos cenários de configuração estática, a fim de que a estimativa de custos fosse possível.



(a) Cenário sem falha.

(b) Cenário de falha CF_1.

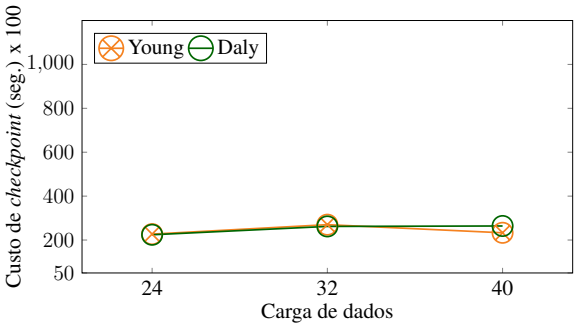
(c) Cenário de falha CF_2.

(d) Cenário de falha CF_3.

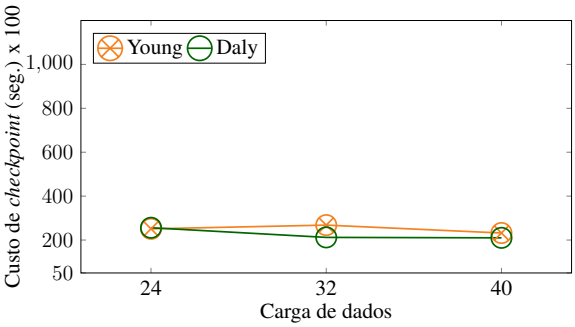
Figura 4.3 – Custo de *checkpoint* com períodos estaticamente configurados.

Com configurações estáticas, é evidente que o custo diminui conforme a frequência de salvamentos aumenta. Com um tempo menor entre salvamentos, há menos operações no `EditLog` e, consequentemente, menos trabalho para atualizar-se o `FSImage`. Variando-se os cenários de falha, os valores de custo para uma mesma frequência de *checkpoint* não possuem grandes diferenças. Com exceção da configuração estática padrão (3600 segundos), há uma tendência de custos maiores com mais falhas.

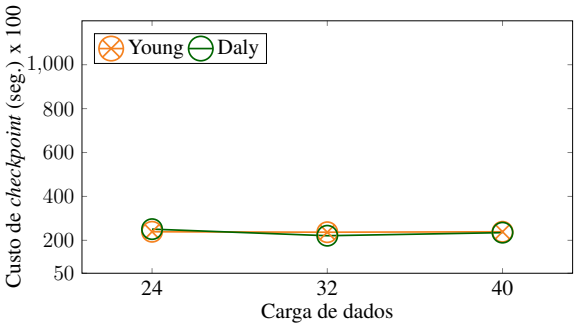
Nos cenários de aplicação do mecanismo dinâmico – conforme gráficos da Figura 4.4 –, os resultados obtidos se mostraram equilibrados, com pouca diferença entre as aproximações usadas. A média do custo de salvamento para ambas as fórmulas se manteve entre 20000 e 25000 segundos, mesmo com variações de carga e de CF. É importante ressaltar que o mecanismo dinâmico não foi executado em cenários sem falha, devido à falta de informações *a-priori* do MTBF.



(a) Cenário de falha CF_1.



(b) Cenário de falha CF_2.



(c) Cenário de falha CF_3.

Figura 4.4 – Custo de *checkpoint* com períodos dinamicamente configurados.

Mesmo que a diferença da variação de custo no mecanismo dinâmico (para uma mesma carga de dados) seja menos evidente que no cenário estático, uma quantidade maior de falhas minimiza os custos. Com falhas mais frequentes, o MTBF estimado tende a ser menor e, por isso, as aproximações de Young e Daly inclinam-se para a escolha de períodos mais curtos. Esse resultado demonstra como o mecanismo dinâmico consegue tempos de execução satisfatórios mesmo com a indução de falhas: o custo de *checkpoints* tende a ser equilibrado para que a recuperação e a intrusividade não sejam custosas.

Os resultados também mostram como o custo de *checkpoints* e o fator MTBF variam no decorrer do tempo, por conta do dinamismo das execuções quanto ao período entre *checkpoints* e aos cenários de falha. Em razão disso, mudanças de contexto são constantemente observadas. Esta característica é importante posto que a arquitetura de configuração dinâmica mantém-se, então, devidamente atualizada e consciente do estado de execução do ambiente computacional. Além disso, a consciência do estado de execução em tempo real é crucial para a definição dos fatores usados quando as informações *a-priori* são desconhecidas.

4.2.4 Abordagem de Aprendizado do Histórico

Como forma de gerar cenários otimizados para o histório, foi criada a abordagem de aprendizado. Nesse cenário, a configuração de *checkpoint* inicial e o conteúdo inicial do histórico são definidos pela base de dados gerada por execuções da primeira abordagem (execuções com base de dados DS_{Empty}). A Figura 4.5 ilustra como os cenários de falha CF_2 e CF_3 são usados quando três cenários de falha (CF_1, CF_2 e CF_3) são definidos. O cenário CF_2 inicia com a base de dados criada pelo cenário CF_1 (DS_1), obtendo-se um cenário CF_2^1 e uma base $DS_{2.1}$. Já o cenário CF_3 baseia-se em DS_1 para obter CF_3^1 e na base $DS_{2.1}$ para obter o cenário CF_3^2.

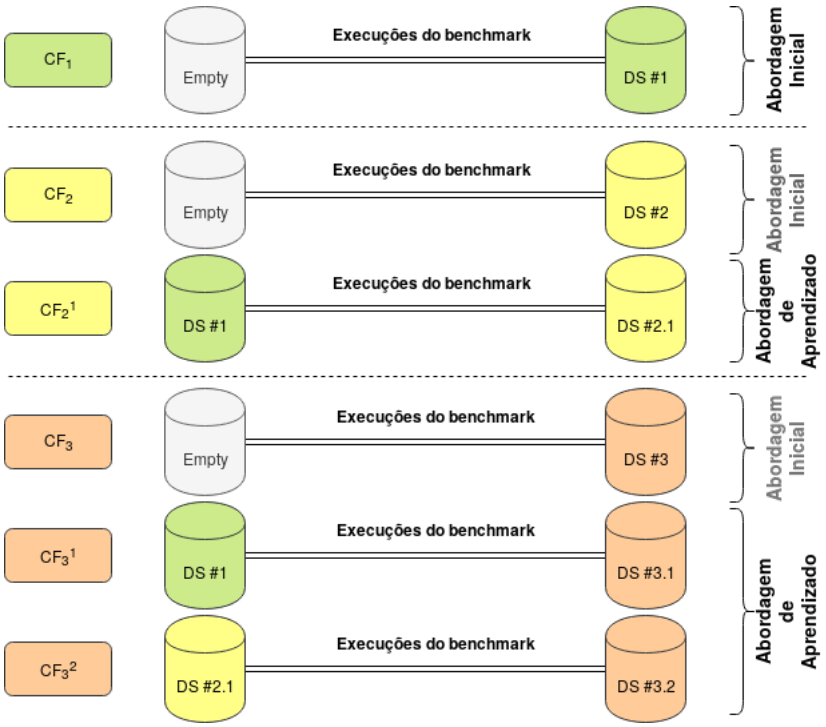


Figura 4.5 – Abordagem inicial e abordagem de aprendizado do histórico de atributos.

A abordagem de aprendizado usa os registros do histórico já estabilizados por execuções da primeira abordagem de execução para otimizar os resultados alcançados através de experiências anteriores. O objetivo é apurar o comportamento do mecanismo de configuração dinâmica em um cenário onde já existem valores observados. A Figura 4.6 exibe o tempo de execução dos cenários testados a partir da base de dados usada.

Na maior parte das execuções baseadas em DS_1, existe uma diminuição no tempo de execução. Esse comportamento se dá pelo fato do *framework* possuir informações sobre os custos de *checkpoints* e sobre as falhas no $NameNode$ desde o início da execução. Ainda, a

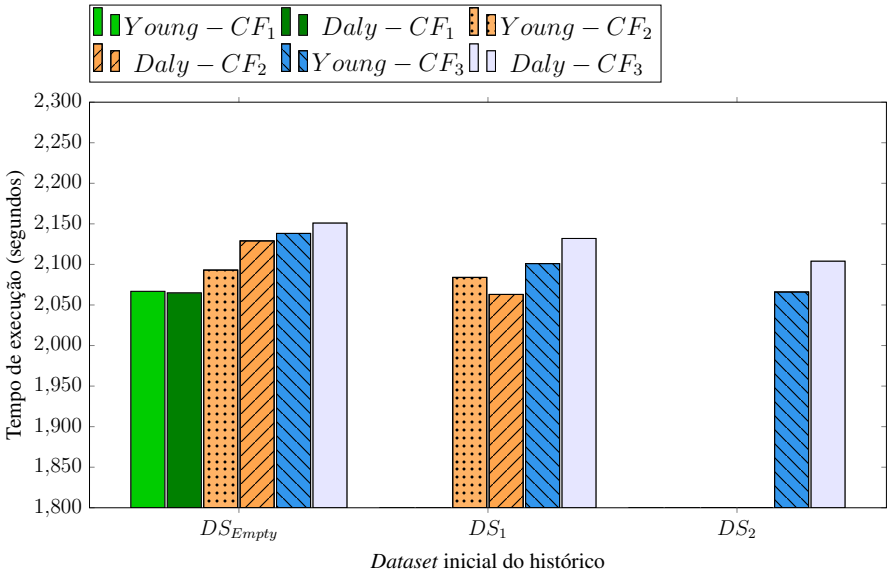


Figura 4.6 – Tempo de execução dos cenários com *datasets* do histórico com a DCA.

inicialização das execuções com um histórico de atributos consolidado (e um período de *checkpoint* mais próximo ao ideal neste início) ajudou no desempenho, mesmo frente a um cenário de falhas diferente. Ou seja, foram necessárias menos execuções para que o histórico tenha adaptado-se novamente ao contexto.

Além disso, a inicialização com um histórico já estabilizado facilita a adaptação ao período ideal com menos execuções. Isto é, execuções no cenário de falhas CF_3 foram as mais beneficiadas pelo uso do histórico de atributos. A base de dados DS_2 foi consolidada com uma série de dados coletados por 40 execuções através de 2 execuções diferentes – e 2 cenários de falhas diferentes. Por isso, a adaptação do período aconteceu de maneira mais rápida, de modo que o tempo de execução obteve um desempenho semelhante ao obtido no *baseline* sem falha.

Os resultados mostraram que o emprego das aproximações de Young e Daly como métricas de monitoramento contribuíram de forma direta para a definição de períodos de *checkpoint* eficientes. Ou seja, observou-se que a recuperação aconteceu de maneira satisfatória mesmo com uma quantidade expressivamente menor de salvamentos (comparada aos cenários estáticos de recuperação eficiente).

Já entre as duas aproximações, percebeu-se uma pequena diferença de desempenho que impossibilita a definição de uma aproximação mais adequada que a outra para os casos testados (conforme Figura 4.6). Em geral, as duas fórmulas obtiveram sucesso. A fórmula de Daly apresenta-se como uma otimização da fórmula de Young, mas suas vantagens não foram observadas de forma impactante devido às características dos testes induzidos por este trabalho.

Testes mais variados ou simulações de ambientes reais de produção podem ajudar a diferenciar o desempenho das duas aproximações.

Para avaliar o desempenho do histórico em diferentes circunstâncias, foram executadas rodadas de teste com duas opções alternativas de tamanho do histórico. As alternativas usaram a metade da capacidade dos primeiros testes (10 entradas) e o dobro da capacidade (40 entradas). As Figuras 4.7 e 4.8 exibem os tempos de execução para 10 e 40 entradas, respectivamente.

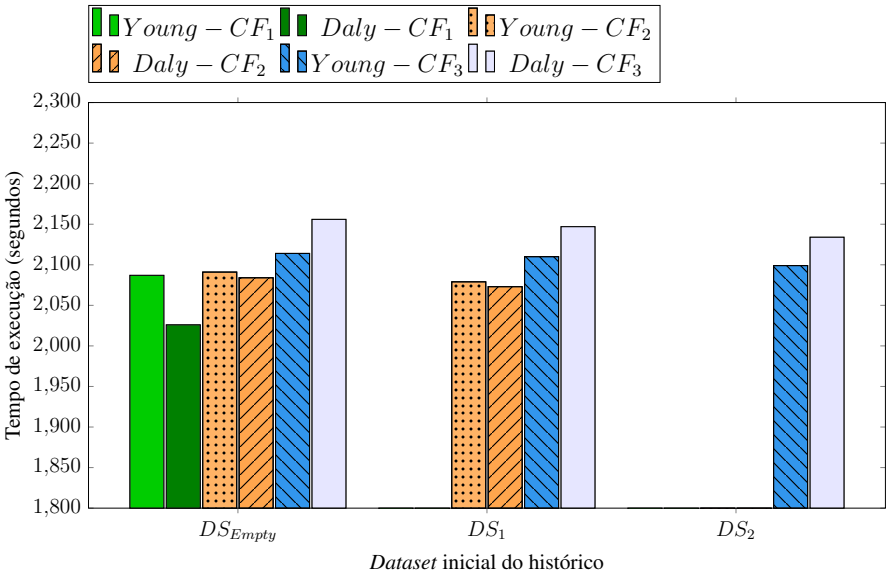


Figura 4.7 – Tempos de execução com 10 entradas no histórico.

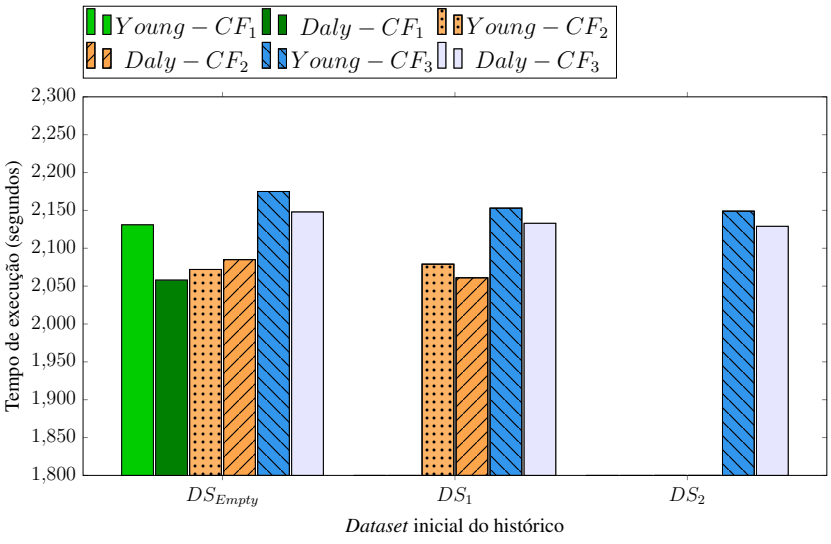


Figura 4.8 – Tempos de execução com 40 entradas no histórico.

Em ambos os casos, não há uma diferença significativa para o histórico com 20 entradas. Esse comportamento mostra que o histórico pode ser eficiente independentemente do tamanho de entradas usadas, desde que exista uma possibilidade de análise dos fatores envolvidos.

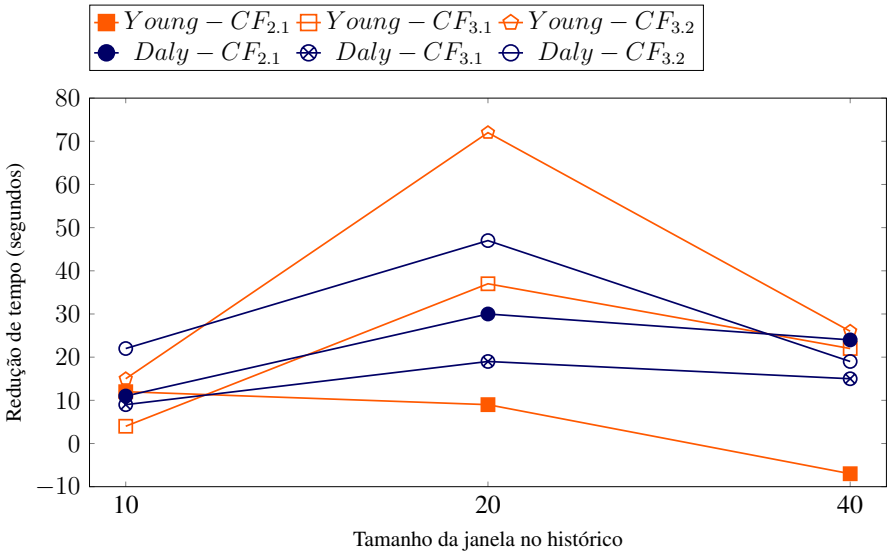


Figura 4.9 – Redução do tempo de execução dos testes a partir da abordagem de aprendizado e dos diferentes tamanhos da janela de observação do histórico.

Com 10 entradas, o histórico apresenta-se ligeiramente mais eficiente nos cenários CF_1 e CF_2. Porém, no cenário CF_3 há um pequeno acréscimo do tempo de execução em comparação com os outros tamanhos de janela de observação. Esse comportamento também se dá com a variação da base de dados usada. Com um comportamento mais aleatório, a análise dos fatores observados é prejudicada por uma visão mais restrita. Por isso, o histórico realiza análises menos fundamentadas em relação aos casos com 20 e 40 entradas.

O comportamento do histórico com 40 entradas é similar ao caso com 20 entradas nos cenários com base de dados DS_{Empty}, visto que ambos trabalham da mesma forma. A diferença se dá apenas nos casos da abordagem de aprendizado. O gráfico da Figura 4.8 mostra que os resultados são favoráveis ao uso de uma visão mais ampla quando a basa de dados inicial já está consolidada, uma vez que os tempos de execução permaneceram no mesmo nível ou menores em relação à primeira abordagem.

Desta forma, destaca-se que uma janela pequena pode ser eficiente para tratar de eventos de falha, mas o aprendizado por outras observações torna-se inviável. Já um cenário com janelas maiores herda a experiência de outras execuções, o que pode ser uma característica importante para que o mecanismo atinja o equilíbrio entre desempenho e confiabilidade mais rapidamente. A Figura 4.9 mostra o tempo que cada cenário da abordagem de aprendizado diminuiu em relação ao seu próprio cenário na abordagem inicial (i.e. with DS_{Empty}).

Em praticamente todos os casos, há uma diminuição no tempo de execução devido à aprendizagem obtida pelo histórico. Nos casos com janela de tamanho 10, há uma menor dis-

crepância entre os cenários, visto que a aprendizagem é pequena. Os melhores resultados são apresentados pelo caso com 20 entradas no histórico, em que a aprendizagem ocorre mais naturalmente. Com 40 entradas, também há uma perspectiva favorável, mas a visão ampla demais em relação ao caso de 20 entradas torna a adaptação ao cenário atual mais lenta.

Em resumo, os resultados obtidos pelos diferentes tamanhos da janela de observação do histórico mostram que o mecanismo de configuração dinâmica pode funcionar de maneira satisfatória em qualquer das variações testadas. Uma janela pequena é útil quando o comportamento de falhas e o custo de *checkpoints* são uniformes. Quando esses fatores possuem uma característica mais aleatória, uma visão abrangente proporciona uma melhor adaptação do período entre *checkpoints* pelo histórico.

4.3 Validação no Spark

A fim de validar a implementação da DCA na tomada de decisão sobre *checkpoints* no Spark, testes de desempenho foram executados nos *benchmarks* PageRank e K-Means. A plataforma HiBench foi usada para que o processo de execução fosse facilitado. Nesse caso, os *clusters* utilizados foram configurados com 24GB de memória RAM para cada nodo do Spark, sendo que apenas 12GB foi destinado para cada *executor* – devido à política de distribuição de memória do *framework*.

Os CFs foram definidos com três cenários baseados no número de falhas por execução: 1 (CF_1), 2 (CF_2) e 3 (CF_3) falhas. O momento de indução das falhas é escolhido de forma pseudo-randômica por rodada de execução, com intervalos baseados nos *baselines* – execuções com política estática *memory-only* (MO) – sem falhas de cada cenário de carga. Isto é, para políticas estáticas ou dinâmicas, a política *baseline* é a MO. A primeira falha ocorre entre 25% e 50%, a segunda entre 50% e 60% e a terceira entre 60% e 75% do tempo a partir do início do *baseline*.

Para facilitar a análise das políticas de persistência, foram definidos RDDs alvo para cada *benchmark*. Esses RDDs consistem em todos os *datasets* do algoritmo que são passíveis de persistência, seja ela estática ou dinâmica. A definição destes *datasets* é essencial quando abordagens estáticas são usadas, pois é necessário conhecer o código do *benchmark* para decidir o tipo de persistência a ser usada em cada *dataset*. Esse problema é corrigido pelo mecanismo dinâmico, que faz as escolhas automáticas.

De modo a validar os métodos de persistência já disponibilizados pelo Spark, foram

definidas as políticas estáticas. Sendo assim, os testes avaliaram as políticas estáticas MO (*memory-only*), MAD (*memory-and-disk*) e CH (*checkpoint*). Ou seja, os RDDs alvo foram marcados para persistência de acordo com a política definida. Na abordagem dinâmica, foram usadas as duas políticas dinâmicas propostas pela arquitetura: *Lineage Threshold* (LT) e *Failure Awareness* (FA). Nesse cenário, os RDDs alvo não são necessários – já que as políticas decidem os RDDs a serem persistidos de forma automática –, mas são usados como base para comparações.

A política de *threshold* de *lineage* foi testada com limiares definidos em 3 (LT_3) e 10 (LT_{10}) dependências. A ideia dos *thresholds* definidos é testar a política em dois casos distintos. O caso LT_3 foi proposto para validar um cenário mais rígido, em que a maior parte dos RDDs deve ser considerada para *checkpoint*. Com 10 dependências, o intuito é validar um cenário cujos RDDs iniciais são descartados, de modo que apenas RDDs com *lineage* mais longo são considerados. Testes com *thresholds* diferentes serão considerados em trabalhos futuros, para que a validação torne-se completa para a política LT.

4.3.1 PageRank

A implementação do PageRank no Spark divide-se em três etapas principais. Na primeira etapa, os arquivos são carregados do seu local de origem e transformados em um *dataset* contendo um par chave-valor, que consiste na referência a uma página e sua pontuação inicial. Neste passo, atribui-se o valor de $1/N$ para todas as N entradas do arquivo. Desta forma, o RDD $ranks$ é estabelecido. Em caso de persistência, esse RDD deve ser priorizado: caso uma recomputação seja necessária, mantê-lo em memória evita acessos a sistemas externos – como o HDFS.

A segunda etapa do algoritmo consiste nas iterações necessárias até chegar ao resultado final. Uma iteração consiste em alterar o conteúdo do RDD $ranks$[3] com novas pontuações calculadas. O cálculo das novas pontuações sempre leva em consideração os valores anteriores, sendo que há uma forte dependência entre os *datasets* ao longo das iterações. Contudo, nenhuma ação é realizada durante esta etapa. Sendo assim, todas as iterações são feitas em um único *job* chamado na terceira etapa, em que os resultados são reunidos e salvos de volta no HDFS.

[3] Um novo RDD é criado, com referência ao anterior, já que o conteúdo de um RDD não pode ser efetivamente alterado.

Dessa forma, o algoritmo PageRank no Spark foi validado através de testes de desempenho com variação das políticas de persistência, nos cenários estáticos, e com variação das políticas de *checkpoint* dinâmico quando a arquitetura proposta foi usada. O *framework* HiBench foi usado como plataforma de execução do algoritmo. A configuração das execuções considerou 4 aspectos: (a) a quantidade de páginas presentes no arquivo de entrada, (b) o número de iterações feitas pelo *benchmark*, (c) os RDDs alvo para persistência e (d) o cenário de falhas.

Como forma de validar o desempenho e a recuperação dos dados em eventos de falha, foram testados 3 cenários de falha (CF_1, CF_2 e CF_3), além de execuções sem falhas. Além disso, os RDDs $ranks$ e $result$ – criados antes e depois das iterações, respectivamente – foram definidos como alvos. O RDD $ranks$ consiste no mapeamento dos *links* do arquivo de entrada para a estrutura do Spark. Já o RDD $result$ armazena o rankeamento final após o processamento do algoritmo.

4.3.1.1 Políticas Estáticas

Os resultados obtidos para os testes com políticas estáticas foram sumarizados pela Tabela 4.4 através do tempo de execução e do desvio padrão obtido para cada variação do CF e do número de iterações. As políticas estáticas *memory-only* (MO), *memory-and-disk* (MAD) e *checkpoint* (CH) foram usadas. Neste caso, foi usada uma entrada de dados com 1 milhão de *links*. De forma semelhante, a Tabela 4.5 mostra os resultados obtidos pelos testes com 5 milhões de *links* como entrada. Os cenários de 1 e 5 milhões usam um arquivo do HDFS de 2,4GB e 12GB cada.

Conforme mostram os resultados das Tabelas 4.5 e 4.4, é possível verificar que cenários sem falha indicam uma vantagem na utilização da política de armazenamento em memória, devido ao rápido acesso aos *datasets* persistidos. À medida que o disco é usado via *checkpoint*, o desempenho tende a ser menos eficiente no cenário sem falha.

A política MAD mostrou desempenho idêntico à política MO em ambos os cenários de carga, uma vez que a quantidade de dados envolvida não justifica o uso do disco pela política. Por outro lado, a política CH apresentou um tempo de execução maior em cenários sem falhas. Ainda assim, a diferença entre as políticas é pequena dado que as políticas MO e MAD possuem a vantagem da persistência em memória.

Esse comportamento deve-se ao fato da implementação do PageRank não efetuar mais

Política	Cenário de Falha	Iterações					
		5		10		15	
		Execução (s)	Dv. Padrão (s)	Execução (s)	Dv. Padrão (s)	Execução (s)	Dv. Padrão (s)
	S/Falha	113,4	6,4	122,1	6,4	141,1	7,1
MO	CF_1	153,8	11,3	164,5	11,8	184,4	8,5
	CF_2	160,7	9,8	183,8	11,8	201,0	16,9
	CF_3	163,6	22,4	196,9	19,4	206,5	19,5
	S/Falha	112,1	8,5	121,5	7,6	143,5	7,0
MAD	CF_1	152,4	10,6	164,5	12,6	185,2	9,6
	CF_2	161,0	9,5	178,8	10,1	200,9	10,4
	CF_3	166,6	18,7	199,1	12,3	204,0	14,9
	S/Falha	132,5	10,1	157,8	9,8	170,2	10,3
CH	CF_1	148,7	10,1	177,8	10,6	189,5	12,4
	CF_2	152,7	10,6	182,9	12,6	194,4	11,6
	CF_3	161,2	8,4	184,1	9,5	200,1	10,1

Tabela 4.4 – Resultados obtidos pelas políticas de persistência estáticas do Spark no PageRank com 1 milhão de páginas.

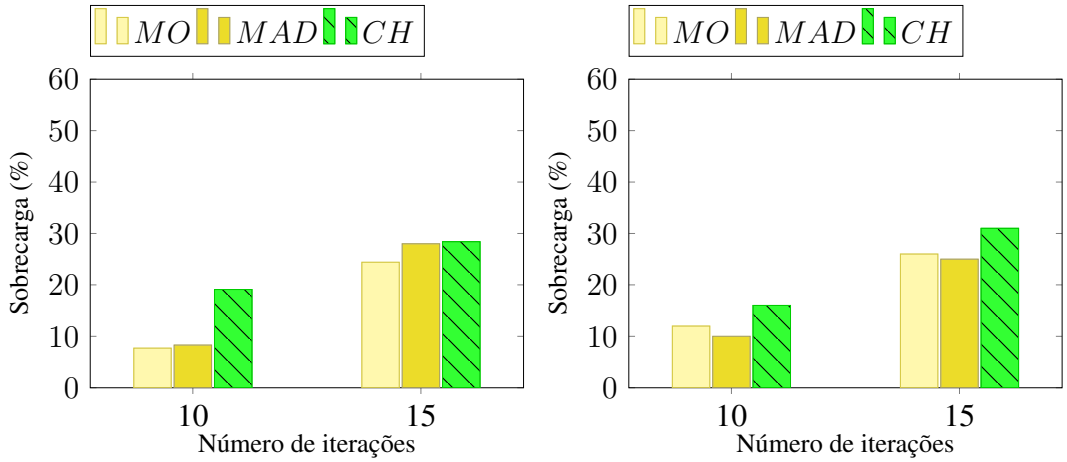
Política	Cenário de Falha	Iterações					
		5		10		15	
		Execução (s)	Dv. Padrão (s)	Execução (s)	Dv. Padrão (s)	Execução (s)	Dv. Padrão (s)
	S/Falha	894,1	36,4	997,0	35,1	1122,1	37,4
MO	CF_1	1191,0	62,4	1290,4	64,5	1450,0	65,0
	CF_2	1381,2	56,9	1502,3	50,8	1690,8	62,1
	CF_3	1588,9	42,6	1971,0	61,4	2071,9	68,6
	S/Falha	896,5	24,5	986,8	39,0	1120,4	21,4
MAD	CF_1	1231,5	54,1	1388,4	56,0	1572,6	41,0
	CF_2	1372,4	39,0	1577,8	56,0	1888,7	51,0
	CF_3	1588,4	50,0	1999,2	51,0	2198,5	55,0
	S/Falha	964,9	33,0	1114,3	80,3	1258,7	64,1
CH	CF_1	1148,1	59,4	1336,9	84,6	1567,6	102,3
	CF_2	1252,5	150,9	1423,2	148,6	1647,9	128,7
	CF_3	1335,1	125,9	1562,9	119,5	1731,7	128,2

Tabela 4.5 – Resultados obtidos pelas políticas de persistência estáticas do Spark no PageRank com 5 milhões de páginas.

do que uma leitura do *dataset* salvo em *checkpoint*, independentemente do número de iterações. Ou seja, apenas um *job* (ação) é realizado ao final de todas as iterações, de modo que a leitura de dados do HDFS acontece uma única vez e os dados permanecem na memória de execução do *driver* até o final do *job*.

Por consequência, o reuso de dados não limita o desempenho da execução quando o *dataset* é salvo em *checkpoint*. As sobrecargas de iteração, apresentadas na Figura 4.10, indicam o tempo extra de processamento com 10 e 15 iterações em relação ao teste com 5 iterações (*baseline*) nos cenários de 1 e 5 milhões.

Em cenários sem falha, o comportamento do *checkpoint* mostrou-se pouco intrusivo mesmo que tenha adicionado o tempo de leitura em disco. Esperava-se que a diferença do tempo de execução nos testes com *checkpoint* aumentasse com mais iterações, em comparação

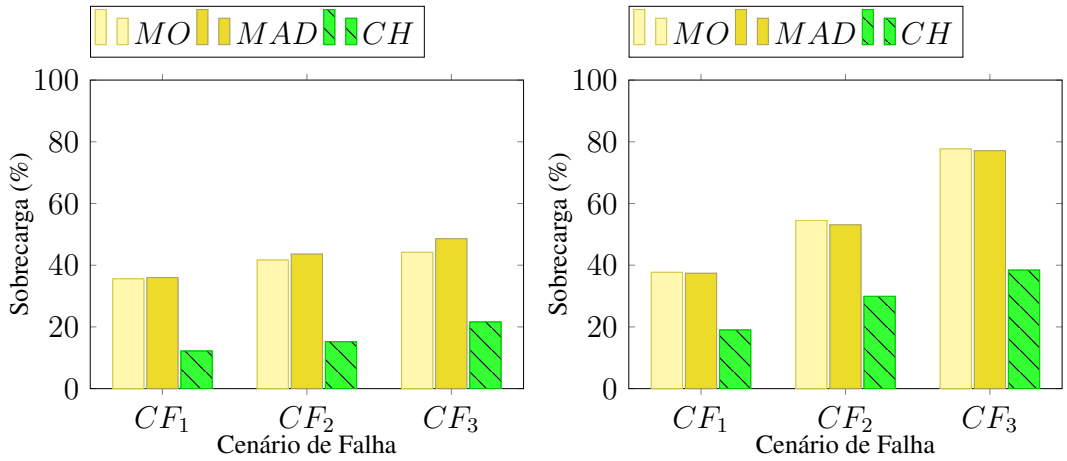


(a) Cenário com 1 milhão de páginas. (b) Cenário com 5 milhões de páginas.

Figura 4.10 – Sobrecargas de iteração geradas por cenários de teste do PageRank com políticas estáticas.

com a política MO. Porém, a sobrecarga de iterações apresentada manteve um nível semelhante nas três políticas estáticas.

Os resultados da política de *checkpoint* tornam-se mais favoráveis quando eventos de falha são observados no sistema. Pelos resultados das Tabelas 4.4 e 4.5, é possível verificar que o número de falhas é altamente intrusivo quando a aplicação atua com a política MO. Com a política CH, a sobrecarga gerada é substancialmente menor. A Figura 4.11 mostra o tempo excedente de execução com 5 iterações nos cenários de falha, em relação ao cenário sem falha, a partir da sobrecarga de CF.



(a) Cenário com 1 milhão de páginas. (b) Cenário com 5 milhões de páginas.

Figura 4.11 – Sobrecargas de CF geradas por cenários de teste do PageRank com políticas estáticas.

As políticas estáticas com uso de memória apresentaram um alto nível de sobrecarga de

execução criado pelas situações de recuperação pós-falha: primeiramente, o reprocessamento das tarefas incompletas e das partições perdidas é feito; depois, a execução dos *jobs* com a ausência de um ou mais *workers* naturalmente implica em uma sobrecarga de execução. Porém, os reprocessamentos atrasam o início de novas tarefas, o que incide em três fatores de atraso: as recomputações, o atraso para início de novas tarefas e a ausência de *workers* falhos.

No caso de RDDs salvos em *checkpoint*, a situação de reprocessamento é substituída pela sobrecarga de leitura, que demandou um tempo menor. Ou seja, assim que uma falha é detectada, não há necessidade de reprocessamento das partições perdidas. Ao invés, o Spark busca as partições falhas no repositório usado pelo *checkpoint* e evita o atraso de recomputação para reiniciar as tarefas perdidas. O aumento no número de falhas simultâneas por execução mostrou de forma mais clara a melhora no desempenho de recuperação de *datasets* perdidos quando salvos em *checkpoint*.

Assim como o fator reutilização, a baixa carga de dados envolvida não prejudicou o desempenho da política CH. Por isso o procedimento de salvamento apresentou-se pouco custoso. A Tabela 4.6 exibe o tempo gasto com o salvamento de cada RDD alvo no cenário sem falha, além do percentual de carga que esse tempo representa no tempo de execução final. O tempo de salvamento nos CFs manteve-se idêntico, uma vez que a presença de falhas no Spark não interfere no desempenho de escrita do HDFS.

Carga de dados	***Dataset***	**T. Salvamento**	**Sobrecarga**	**Qtd. Salvamentos**
1mi	$points$	1,87 segundos	1,41%	1
	$result$	0,6 segundos	0,45%	1
	Total	**2,47 segundos**	**1,86%**	**2**
5mi	$points$	54,25 segundos	5,62%	1
	$result$	2 segundos	0,21%	1
	Total	**56,25 segundos**	**5,83%**	**2**

Tabela 4.6 – Quantidade e tempo de salvamento dos *checkpoints* com a política estática CH nos cenários sem falha e 5 iterações do PageRank.

O salvamento de *checkpoint* gera pouca sobrecarga no tempo de execução quando realizado de forma única (isto é, quando não há mais de um salvamento por RDD). A baixa sobrecarga é observada pois o *job* de *checkpoint* é executado tão logo o RDD é processado por seu *job*, de modo que os dados já estão em memória. Assim, evita-se a necessidade de um novo processamento. Neste cenário de indução manual, os RDDs submetidos a *checkpoint* foram setados para persistência em memória. Porém, casos em que *checkpoints* são induzidos sem a persistência prévia podem gerar um gargalo de desempenho.

Portanto, é possível observar que a abordagem de *checkpoint* cumpre com suas expectativas relacionadas à rápida recuperação, desde que o repositório seguro escolhido não gere (por si só) um atraso de leitura e escrita. A partir da rápida recuperação em um cenário de falha, a política pode ser uma solução em ambientes pouco confiáveis. Contudo, sua aplicação em cenários sem falha deve ser evitada, já que nenhum procedimento de recuperação é feito. Embora a sobrecarga gerada seja pequena em relação às sobrecargas de CF e iteração, o aumento da carga de dados mostrou uma baixa escalabilidade da política no cenário sem falha. Nesse caso, o salvamento em *checkpoint* torna-se um meio dispensável de persistência.

4.3.1.2 Políticas Dinâmicas

Nos testes com políticas dinâmicas, os RDDs alvo não são necessários – já que as políticas decidem os RDDs a serem persistidos de forma automática –, mas são usados como base para comparações. Os testes foram feitos através das políticas LT, com *thresholds* de 3 (LT_3) e 10 (LT_{10}) dependências, e da política FA. Os resultados obtidos por políticas dinâmicas no cenário com 1 milhão de páginas foram sumarizados pela Tabela 4.7, enquanto a Tabela 4.8 apresenta os resultados para o cenário com 5 milhões de páginas.

Política	**Cenário de Falha**	**Iterações**					
		5		**10**		**15**	
		Execução (s)	**Dv. Padrão (s)**	**Execução (s)**	**Dv. Padrão (s)**	**Execução (s)**	**Dv. Padrão (s)**
LT_3	S/Falha	131,8	10,5	155,6	10,1	172,8	10,0
	CF_1	150,7	12,0	177,1	10,5	189,5	11,0
	CF_2	153,4	10,6	182,6	11,0	195,3	9,1
	CF_3	160,5	11,0	185,5	9,1	199,7	9,2
LT_{10}	S/Falha	118,9	8,4	125,9	8,2	146,5	8,0
	CF_1	156,5	9,5	166,5	9,5	184,3	7,4
	CF_2	161,0	10,2	183,4	11,0	202,0	11,1
	CF_3	166,8	10,9	202,0	10,6	206,7	11,2
FA	S/Falha	115,8	8,3	121,3	8,4	140,1	9,5
	CF_1	150,0	13,5	161,2	12,8	185,6	13,6
	CF_2	154,2	13,6	168,7	7,7	189,6	10,8
	CF_3	162,8	8,4	181,4	9,5	199,5	11,8

Tabela 4.7 – Resultados obtidos pelas políticas dinâmicas de *checkpoint* do Spark no PageRank com 1 milhão de páginas.

Os resultados obtidos pelas políticas dinâmicas alcançaram resultados próximos aos cenários estáticos. No caso da abordagem de *Lineage Threshold* (LT), percebe-se como a instrução automática de *checkpoints* pode diminuir o tempo de execução em alguns casos (LT_3). Mas o estabelecimento ainda estático de um *threshold* não garante um comportamento definitivo para a política, o que implica em variações de desempenho. Desta forma, o uso de políticas

Política	Cenário de Falha	Iterações					
		5		10		15	
		Execução (s)	Dv. Padrão (s)	Execução (s)	Dv. Padrão (s)	Execução (s)	Dv. Padrão (s)
LT_3	S/Falha	968,5	52,5	1111,3	43,0	1279,0	55,7
	CF_1	1156,5	60,4	1336,1	89,6	1555,8	98,5
	CF_2	1272,8	78,5	1444,2	101,0	1697,0	88,2
	CF_3	1398,0	121,4	1567,4	110,7	1712,1	119,9
LT_{10}	S/Falha	1002,5	31,6	1086,1	40,9	1145,0	37,4
	CF_1	1239,0	75,6	1407,0	90,4	1501,0	90,0
	CF_2	1432,3	100,0	1590,2	100,1	1894,5	97,2
	CF_3	1618,5	89,7	1980,6	106,7	2241,3	83,2
FA	S/Falha	904,9	33,0	1024,3	20,3	1078,7	24,2
	CF_1	1327,9	75,6	1407,0	90,4	1894,5	97,2
	CF_2	1422,4	100,0	1590,2	100,6	1894,5	97,2
	CF_3	1618,5	89,7	1980,6	106,7	2241,3	83,2

Tabela 4.8 – Resultados obtidos pelas políticas dinâmicas de *checkpoint* do Spark no PageRank com 5 milhões de páginas.

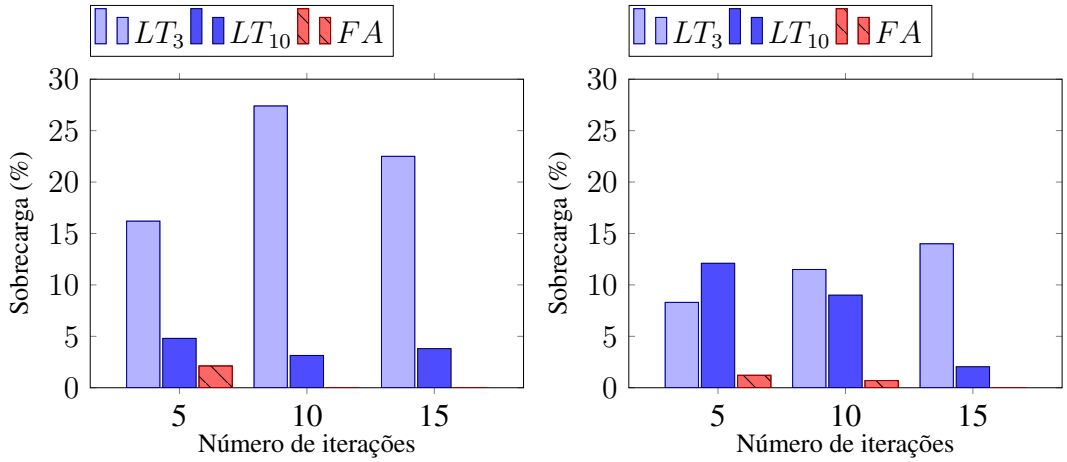
dinâmicas com atributos estáticos mantém o mesmo problema enfrentado pelas políticas estáticas: a escolha por um *threshold* menor facilita cenários de recuperação, mas gera sobrecarga sem falha; quando o *threshold* aumenta, o comportamento é invertido.

A política LT_3 considerou os dois RDDs alvo para o estabelecimento de *checkpoints*. Uma vez que o RDD $links$ é submetido a procedimentos de mapeamento e filtro antes de sua persistência, seu *lineage* é grande o suficiente para o *threshold* de 3 dependências. Nesse caso, a política LT_3 se apresentou de forma semelhante à CH, com tempos de salvamento idênticos.

Já a política LT_{10} submeteu apenas o RDD $results$ para *checkpoint*. Como este RDD é usado apenas no final da aplicação, não há ganho de desempenho ao salvá-lo em *checkpoint*. Por isso, assim como a política MO, um *threshold* de 10 dependências para o PageRank não ofereceu um cenário eficiente de recuperação. Além disso, o tempo gasto com o salvamento do RDD $results$ indica um tempo de execução ainda maior da política LT_{10} frente a política MO.

Por outro lado, a abordagem *Failure Awareness* (FA) não apresentou um comportamento intrusivo em cenários sem falhas, devido ao fator MTBF não induzir a escolha por *checkpoints*. A Figura 4.12 mostra a diferença de desempenho das políticas dinâmicas em comparação com as estáticas através do fator sobrecarga, em cenários sem falha.

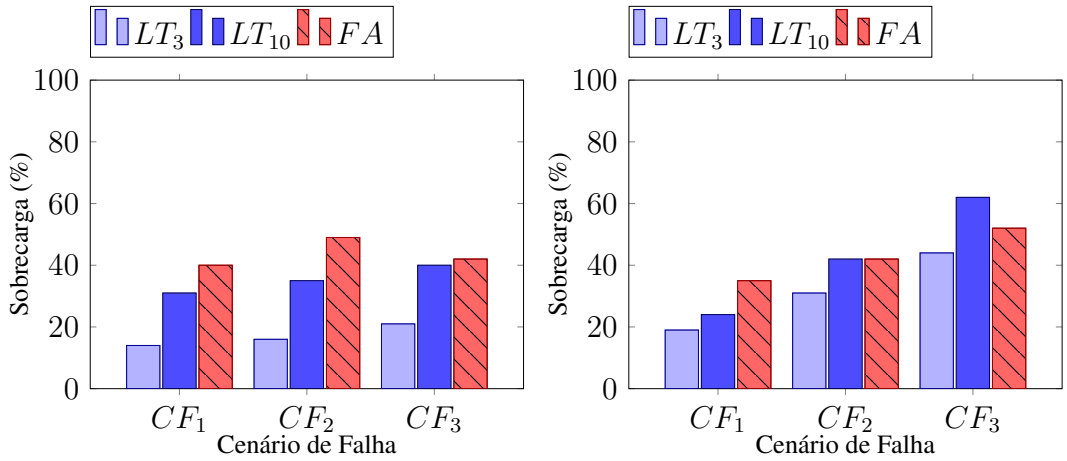
Uma vez que a primeira etapa da política FA é o monitoramento de falhas, seu comportamento em casos sem falhas é idêntico à abordagem de persistência em memória. Sem evidências de falhas constantes, o MTBF tende a aumentar de modo que não seja necessária a análise de custo na etapa 2 da política FA. Conforme o MTBF é alterado nos cenários de falha, a etapa 2 é executada e os cenários de *checkpoint* podem ser observados.



(a) Cenário com 1 milhão de páginas. (b) Cenário com 5 milhões de páginas.

Figura 4.12 – Sobrecargas das políticas dinâmicas no PageRank com 1 e 5 milhões de páginas em relação a política estática MO.

Nos casos com falha, a sobrecarga de CF de execuções com 5 iterações é apresentada pela Figura 4.13. Percebe-se que a menor sobrecarga dentre as políticas dinâmicas é oferecida pela política LT_3. Conforme observado nos testes com políticas estáticas, o salvamento do *dataset* $links$ é determinante para o desempenho de recuperação em CFs. A política LT_3 garantiu o *checkpoint* desse *dataset* em todas as execuções e, por isso, repetiu o bom desempenho da política CH.



(a) Cenário com 1 milhão de páginas. (b) Cenário com 5 milhões de páginas.

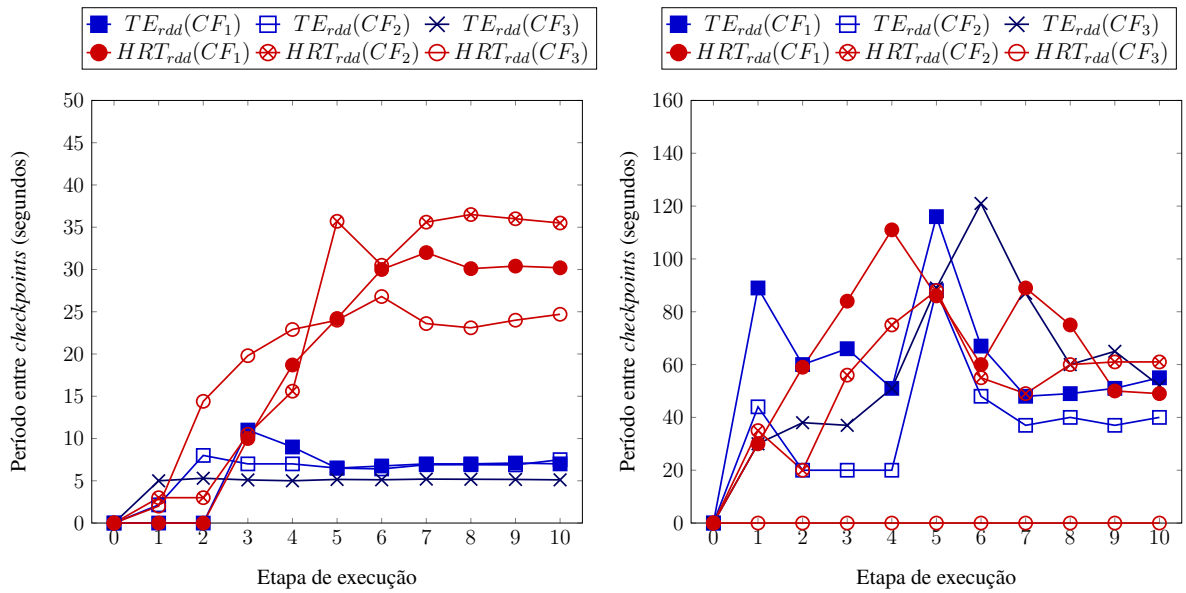
Figura 4.13 – Sobrecargas de CF das políticas dinâmicas no PageRank em 5 iterações com 1 e 5 milhões de páginas.

Assim como foi evidenciado pelos resultados do PageRank com políticas estáticas, a leitura dos blocos salvos em *checkpoint* tornou-se menos custosa que o reprocessamento das partições em cenários de falha em casos com 1 e 5 milhões de *links*. Por outro lado, a política

LT_{10} mostrou um desempenho baseado na política MO, assim como em cenários sem falhas. A falta de *checkpoints* para o *dataset* $links$ torna a recuperação dependente dos procedimentos de recomputação.

Porém, a política FA apresentou uma alta sobrecarga de CF em ambos os cenários de carga. Essa sobrecarga pode ser comparada às sobrecargas de CF das políticas de baixo desempenho em recuperação (MO e LT_{10}). O baixo desempenho da FA nesse fator pode ser entendido pelo seu próprio *baseline*. Como a execução da política sem falhas possui um tempo de execução consideravelmente baixo, a presença de falhas sem a sumbissão estática de *checkpoints* no RDD $links$ em todas as 20 execuções diminuiu o desempenho em cenários de falha.

Esse comportamento também pode ser explicado pelos cálculos realizados para decidir um cenário de *checkpoint*, essencialmente no cenário com 1 milhão de páginas. Por isso, a Figura 4.14 mostra os custos envolvidos na política FA através de etapas de 2 execuções para 1 e 5 milhões de páginas. Os custos são representados pelo tempo de processamento do RDD (TE_{rdd}) e pelo tempo de leitura do RDD em caso de *checkpoint* (HRT_{rdd}).



(a) Cenário com 1 milhão de pontos.

(b) Cenário com 5 milhões de pontos.

Figura 4.14 – Custos de processamento e de leitura no HDFS do RDD $links$ observado pela política FA.

Nos testes com 1 milhão de *links*, a quantidade de *checkpoints* é diminuída e deixa de alcançar 1 *checkpoint* por execução. Como o custo envolvido no processamento dessa quantidade de dados é baixo, a leitura no HDFS se mostrou mais custosa. Por isso, apesar do MTBF

indicar uma probabilidade alta de falha – principalmente no cenário CF_3 –, o fator custo de reprocessamento impede o salvamento de *checkpoints*. Esse comportamento, aliado ao baixo desempenho da política FA nesse cenário de baixa carga de dados, expõe a necessidade de otimizações do cálculo de custo desenvolvido.

Já com 5 milhões, é possível observar pela Figura 4.14 que o tempo estimado para execução dos *jobs* (TE_{rdd}) sobre o *dataset* $links$ possui picos de desempenho. No início da política, em ambos os CFs, a falta de informação sobre falhas atrasa o início da etapa 2 – que faz o cálculo de custos. Por isso, a tendência é de que os primeiros *jobs* não ofereçam a possibilidade de *checkpoints*.

Após o primeiro salvamento, o tempo de execução do *job* com falhas diminui, uma vez que a recuperação via HDFS se mostrou mais rápida que o reprocessamento (vide execuções das políticas LT_3 e CH). Contudo, percebe-se que os custos de processamento são reduzidos à medida que RDDs salvos em *checkpoint* diminuem o tempo de execução. Esse comportamento faz com que os *checkpoints* deixem de ser considerados novamente, gerando um acréscimo no tempo de execução.

O tempo de leitura do HDFS manteve um crescimento semelhante para os 3 cenários de falha dos testes com 1 milhão de páginas. Como a quantidade de *checkpoints* é pequena nesse caso, o tempo de leitura não é afetado por recuperações. Por isso, o uso do HDFS por outras operações molda o tempo médio de leitura igualmente nos CFs. Por outro lado, esse fator é inconstante no cenário de 5 milhões de páginas, acompanhando a maior variação de *checkpoints* realizados e também a variação no tempo de execução dos *jobs*.

O tempo de execução do *job* cresce de acordo com o aumento de carga. Por isso, este cenário apresentou uma maior quantidade de salvamentos em comparação com o cenário de 1 milhão de *links*. Nos dois casos, o *dataset* de resultados ($result$) é considerado pequeno, já que contém apenas informações calculadas e resumidas. Por isso, seu conteúdo deixou de ser salvo em *checkpoint*. Uma vez que uma eventual recuperação pós-falha foi considerada mais rápida que a leitura em disco, a política FA tratou de evitar seu salvamento.

4.3.1.3 Discussão

Os resultados apresentados pelos testes sobre o *benchmarks* PageRank mostram que a escolha por salvamentos em *checkpoint* é eficiente em casos de falha, mas adicionam tempo de execução para escrita e leitura de arquivos em disco. Esse comportamento era esperado

de acordo com a própria definição do *checkpoint* no Spark. Porém, a baixa complexidade de código do PageRank proporciona um cenário de pouca intrusividade em operações de *I/O*. Assim, a política de *checkpoint* torna-se um cenário favorável para a aplicação.

Teoricamente mais eficiente em aplicações *CPU-Intensive*, a política de persistência em memória mostrou-se abaixo do esperado. Apesar de manter os dados em memória para um acesso mais rápido, a pouca reutilização efetiva comprometeu o conceito da política. Uma vez que o reuso é feito dentro de um único *job*, o diferencial da política não foi totalmente explorado. Consequentemente, mesmo as políticas de persistência com uso do disco utilizaram-se da memória de execução durante praticamente todo o *job*. Em cenários de falha, a necessidade de recomputação evidenciou o baixo desempenho da política MO.

Apesar da pequena quantidade de RDDs alvo, observou-se que as abordagens dinâmicas ofereceram uma grande vantagem frente as estáticas. A possibilidade de indução automática de *checkpoint* evitando o acesso ao código é essencial mesmo em aplicações com baixa complexidade de código. Assim, não é necessário que o desenvolvedor da aplicação tenha conhecimento técnico sobre o código para que soluções de *checkpoint* sejam adicionadas ou removidas da execução. Essa perspectiva também evita que o código seja alterado e compilado diversas vezes, sobretudo quando esse tipo de tarefa é difícil ou o acesso ao código fonte não é simples.

De forma mais aprimorada, a política *Failure Awareness* conseguiu o resultado mais expressivo dentre todos os cenários de persistência testados. Com o uso do histórico, foi possível estimar cenários de falha automaticamente. Ainda que os cenários testados sejam de aspecto controlado, a possibilidade de visualizar eventos de falha tornou as execuções mais eficientes.

Com um cenário de maior carga de dados, a política FA alcançou o equilíbrio proposto pela DCA: controle de desempenho sem falhas e controle de confiabilidade em cenários de falha. Ainda que a sobrecarga de CF apresentada pela política não seja a menor dentre as políticas testadas, a união desse resultado com a sobrecarga em relação a política MO evidencia que a FA se apresentou de forma eficiente em ambos os cenários com e sem falha.

Contudo, o cenário de carga com 1 milhão de páginas mostrou que a política FA deixou de considerar o salvamento de *checkpoints* devido ao baixo tempo de execução estimado dos *jobs*. Mesmo com uma execução rápida dos *jobs*, as políticas LT_3 e CH mostraram que o estabelecimento de *checkpoints* é eficiente para esse caso. Já a política FA observou o contrário na maioria da execuções nesse cenário de carga. Por isso, otimizações da política devem ser desenvolvidas para corrigir esse tipo de comportamento.

4.3.2 K-Means

A validação do algoritmo K-Means no Spark foi feita através de testes de desempenho com variação das políticas de persistência, nos cenários estáticos, e com variação das políticas de *checkpoint* dinâmico quando a arquitetura proposta foi usada. Assim como na validação do PageRank, usou-se o *framework* HiBench como plataforma de execução. O arquivo de entrada das execuções contou com execuções a partir de 500 milhões de pontos e 1 bilhão de pontos, totalizando cenários de carga com 48GB e 96GB de dados de entrada, respectivamente. Foram feitas validações com a geração de 10 *clusters* com pontos de 3 dimensões.

A versão do K-Means no HiBench – *framework* usado para as validações – é uma interface para a implementação do algoritmo na biblioteca MLlib do Spark. Por isso, a característica do *benchmark*, em questão de código, o torna significativamente mais complexo que o PageRank. Por possuir um maior número de *datasets* e agrupamentos no *driver*, há uma maior possibilidade de análise conforme políticas de persistência são testadas. Assim, o algoritmo também traz uma quantidade maior de RDDs alvo. A Tabela 4.9 mostra os *datasets* alvo, além do método de persistência padrão usado pelo MLlib, para execuções com n iterações e i etapas de inicialização.

RDD	**Qtd. de Reuso**	**Persistência padrão**
$data$	1	*memory-only*
$norms$	n	*memory-only*
$costs$	i	*memory-and-disk*
$centers$	n	-

Tabela 4.9 – RDDs alvo das execuções do K-Means no Spark.

Dentre os *datasets* da Tabela 4.9, apenas o RDD $data$ pode ser manipulado via código sem que um acesso ao MLlib seja necessário. Esse *dataset* faz ao conjunto de pontos do arquivo de entrada e é passado para o algoritmo assim que o MLlib é chamado. Por sua vez, o RDD $norms$ armazena os dados normalizados e passa a representar os pontos de entrada. Seu conteúdo é persistido em memória pois é utilizado n vezes, sendo n o número de iterações. Ainda que o K-Means utilize o RDD $norms$ durante as iterações, sua complexidade é pequena: apenas um mapeamento o difere do RDD $data$, mantendo-se uma forte dependência entre esses *datasets*.

Na etapa de inicialização, acontece a escolha dos centróides iniciais através de i etapas de inicialização (2 etapas, neste caso). O RDD $costs$ é atualizado i vezes, pois define os pontos

e suas distâncias aos centróides. Por isso, o RDD é persistido com a política *memory-and-disk*. Após a etapa de inicialização, quando o processamento do algoritmo é efetivamente iniciado, o RDD $centers$ surge para armazenar informações sobre os novos centróides. Um novo cálculo é feito a cada iteração, logo esse *dataset* é modificado n vezes.

4.3.2.1 Políticas Estáticas

Os testes estáticos no K-Means foram submetidos com as políticas MO (*memory-only*), MAD (*memory-and-disk*) e CH (*checkpoint*). Nesses testes, é importante ressaltar a limitação apresentada para alterações em código no K-Means. Por isso, as políticas estáticas foram definidas sem alteração nas políticas padrão do MLlib, com exceção do *dataset* $data$ que armazena a entrada inicial de dados da aplicação. Já que o código do K-Means no MLlib é encapsulado e inacessível por usuários do HiBench, alterações no método de persistência do restante dos RDDs alvo não foram possíveis nos testes com políticas estáticas.

O K-Means com políticas estáticas foi executado através de 5, 10 e 15 iterações, além dos 3 CFs definidos para o *benchmark*. Os resultados obtidos são sumarizadas pela Tabela 4.10, para cenário de carga com 500 milhões de pontos, e pela Tabela 4.11 para o cenário com 1 bilhão de pontos. Ambas as Tabelas 4.10 e 4.11 mostram o tempo de execução e o desvio padrão dos diferentes cenários de teste.

Política	**Cenário de Falha**	**Iterações**					
		5		**10**		**15**	
		Execução (s)	**Dv. Padrão (s)**	**Execução (s)**	**Dv. Padrão (s)**	**Execução (s)**	**Dv. Padrão (s)**
	S/Falha	159,2	26,4	175,9	20,7	186,1	36,9
MO	CF_1	290,0	47,0	325,5	41,9	349,15	38,3
	CF_2	395,11	45,9	468,3	61,6	486,5	50,8
	CF_3	476,8	66,5	571,6	79,0	599,4	88,9
	S/Falha	158,9	20,1	177,6	18,4	188,8	28,1
MAD	CF_1	276,0	44,6	349,8	44,1	394,9	40,0
	CF_2	399,4	55,1	451,0	41,9	479,2	61,6
	CF_3	476,7	98,7	552,7	61,2	601,0	78,5
	S/Falha	1598,0	114,9	1999,0	122,7	2518,0	160,1
CH	CF_1	2097,0	241,8	2474,0	365,3	3000,1	249,4
	CF_2	3488,0	299,7	3753,1	301,4	4215,6	300,4
	CF_3	4643,6	457,1	5474,0	397,0	6457,0	500,1

Tabela 4.10 – Resultados obtidos pelas políticas de persistência estáticas do Spark no K-Means com 500 milhões de pontos.

Aplicações como o K-Means exigem uma quantidade considerável de reuso de dados à medida que o número de iterações é maior. Nesse caso, cada iteração exige um agrupamento no *driver* através de uma ação. O armazenamento de dados em memória mostrou-se essencial para

Política	Cenário de Falha	Iterações					
		5		10		15	
		Execução (s)	Dv. Padrão (s)	Execução (s)	Dv. Padrão (s)	Execução (s)	Dv. Padrão (s)
	S/Falha	1232,5	21,4	1589,6	40,6	1670,0	67,1
MO	CF_1	1391,8	66,5	1795,7	61,9	1975,4	74,6
	CF_2	1694,6	66,6	2252,7	74,3	2428,7	71,4
	CF_3	2393,4	104,3	2803,0	126,3	3029,2	114,0
	S/Falha	1170,0	69,1	1426,7	71,6	1542,6	74,7
MAD	CF_1	1426,7	106,4	1875,6	198,7	2279,3	184,0
	CF_2	1775,3	121,9	2275,2	165,4	2468,2	118,5
	CF_3	2312,5	141,5	2956,8	142,9	3110,9	136,6
	S/Falha	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$
CH	CF_1	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$
	CF_2	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$
	CF_3	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$

Tabela 4.11 – Resultados obtidos pelas políticas de persistência estáticas do Spark no K-Means 1 bilhão de pontos.

o desempenho da aplicação em cenários sem falha. Nesses cenários, a diferença de desempenho é evidente graças à disponibilidade em memória dos dados no início de cada iteração.

No cenário de menor carga de dados, o desempenho das políticas MO e MAD mostrou-se idêntico, já que não houve uma sobrecarga de dados suficiente para despejo de *datasets* da memória para o disco. Já no cenário com maior carga de dados, uma parte dos *datasets* armazenados foi persistido em disco pela política MAD, refletindo em um pequeno ganho de desempenho em relação à MO. Como a política MO precisa lidar com reprocessamento de *datasets* que não cabem em memória, o disco apresentou-se como uma solução mais eficiente.

Por outro lado, os testes com a política de *checkpoint* (CH) ressaltaram a perda excessiva de desempenho causada pelo acesso frequente ao disco pelo *benchmark*. O tempo de execução aumenta consideravelmente quando apenas o *dataset* $data$ é persistido em *checkpoint*. Nos casos com 1 bilhão de pontos, inclusive, os testes não foram completados devido ao tempo de execução excessivo. Por isso, a Tabela 4.11 traz valores $undefined$ para o resultado dos testes nesse cenário.

O alto nível de intrusividade da política CH pode ser melhor entendido pelo custo de salvamento envolvido. A Tabela 4.12 exibe o tempo de salvamento do RDD $data$ em *checkpoint* nos diferentes cenários de teste, além do percentual que esse valor representa no tempo de execução obtido (em %).

Nota-se que o custo de salvamento é um fator determinante para o desempenho do *benchmark* sob a política CH. O alto custo é justificado pelo número de partições criadas para o *dataset*. Com as configurações de execução usadas, o *dataset* $data$ foi dividido em 384 e 768

Cenário de Falha	T. Salvamento (s)	Sobrecarga
S/Falha	337,9	22,3%
CF_1	348,4	16,6%
CF_2	378,5	10,8%
CF_3	358,0	6,6%

Tabela 4.12 – Quantidade e tempo de salvamento de *checkpoint* do *dataset* $data$ com a política estática CH nos cenários de 5 iterações.

partições para 500 milhões e 1 bilhão de pontos, respectivamente. Ou seja, uma partição foi criada para cada 128MB de entrada. Esse tamanho de partição representa o tamanho padrão de um bloco do HDFS. Como os dados de entrada são carregados do HDFS, o número de partições no Spark também é definido de acordo com o número de blocos do arquivo.

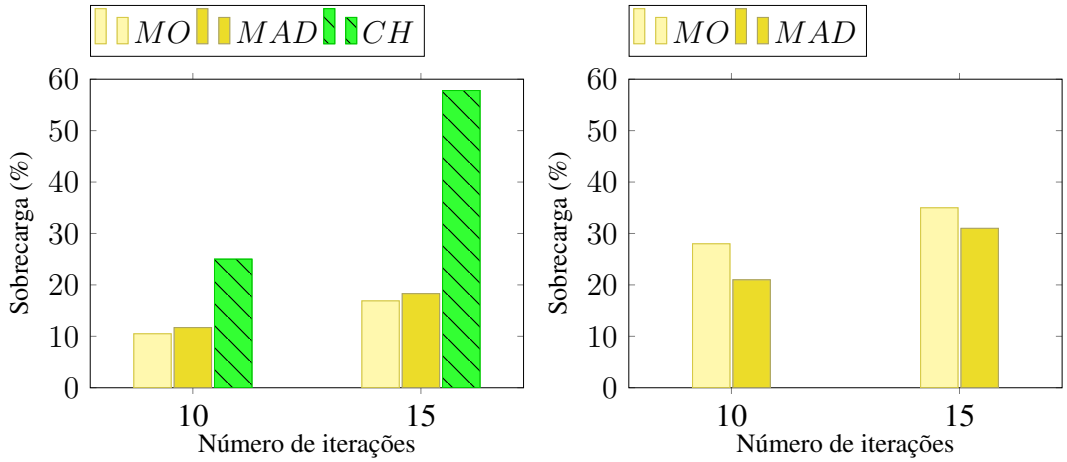
Ao dividir o arquivo de entrada em diversos blocos, o desempenho do HDFS tende a ser pior que a divisão dessas partições por *executors* no Spark. Assim, o desempenho de processamento do Spark tende a ser melhor que o desempenho de *I/O* do HDFS com dados mais particionados. Porém, a sobrecarga de salvamento é baixa, de modo que o salvamento por si só não reflete a maior parte do tempo de execução.

Em cenários de falha, esse comportamento é melhor observado já que a sobrecarga do salvamento em relação ao tempo de execução é menor conforme mais falhas são induzidas. Além do salvamento, então, é possível notar que a leitura de uma grande quantidade de partições do HDFS também mostrou-se custosa e intrusiva, justificando a persistência em memória.

A característica do K-Means é importante para o entendimento dos resultados obtidos pela política CH. Como o *dataset* $data$ é grande – pois contém todos os pontos de entrada –, sua leitura é custosa. Ademais, sua leitura é requisitada em diversos momentos da aplicação, como nas n iterações e nas i etapas de inicialização dos pontos. Desta forma, a grande repetição de leituras custosas justifica a sobrecarga excessiva da política.

A discrepância de desempenho entre as políticas MO e CH é ainda mais relevante quando o número de iterações aumenta. A Figura 4.15 exibe a sobrecarga de iterações observada em cada cenário de teste sem falha. O *baseline* da sobrecarga de iterações é o caso com 5 iterações de cada execução. No caso com 1 bilhão de pontos, não foi possível calcular sobrecargas devido à falta de informação sobre o tempo de execução.

No caso de 1 bilhão de dados, a quantidade de dados maior que a capacidade de memória dos *executors* mostrou que a política MAD garantiu uma sobrecarga menor entre iterações. Já com 500 milhões de pontos, é possível notar que a política de *checkpoint* sofre uma queda de

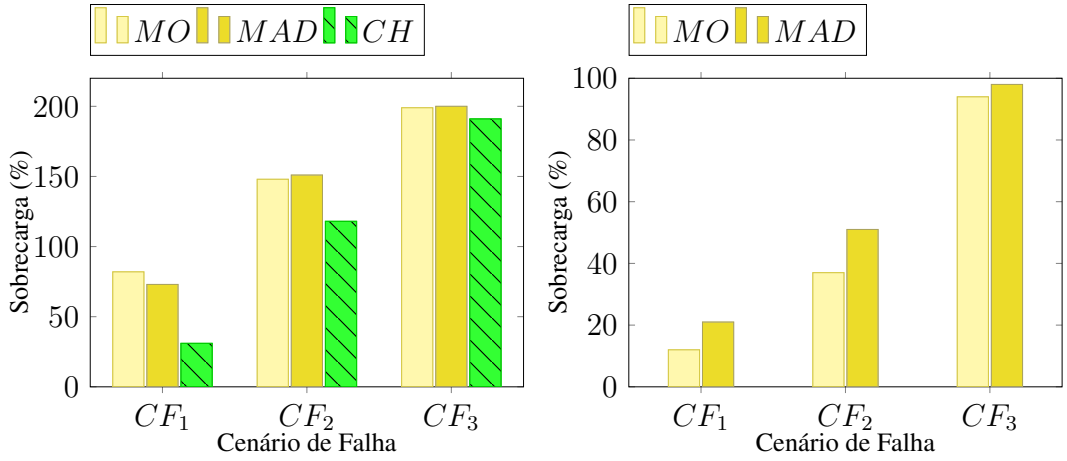


(a) Cenário com 500 milhões de pontos. (b) Cenário com 1 bilhão de pontos.

Figura 4.15 – Sobrecargas de iteração geradas por cenários sem falha do K-Means com políticas estáticas.

desempenho quando mais leituras do disco são necessárias. Com os dados em memória, as iterações tendem a gerar pouca intrusividade, desde que recomputações não sejam necessárias.

Por outro lado, percebe-se que o cenário de *checkpoint* mostra um tempo de recuperação mais eficiente em cenários de falha. Isto é, a diferença de tempo de execução entre cenários de falha da política CH é a menor em comparação com as demais políticas estáticas. A Figura 4.16 destaca essa observação através das sobrecargas de CF obtidas. Assim como nas comparações anteriores, a análise da sobrecarga na política CH com 1 bilhão de pontos foi impossibilitada.



(a) Cenário com 500 milhões de pontos. (b) Cenário com 1 bilhão de pontos.

Figura 4.16 – Sobrecargas de CF geradas por cenários de 5 iterações do K-Means com políticas estáticas.

A sobrecarga de CF apresenta-se menor nos cenários 1 bilhão de pontos, mesmo tratando-se de um caso com o dobro de dados de entrada. A diferença se dá pelo fato do cenário *baseline*

(sem falha) ser mais eficiente com 500 milhões de pontos. Uma vez que a carga de dados é menor, o espaço em memória é melhor aproveitado pelas políticas. Com uma limitação de espaço, mesmo os cenários sem o fator prejudicial de falhas são afetados.

Em cenários de falhas, o desempenho das políticas estáticas é definido pelo comportamento das etapas de recuperação. As sobrecargas de CF exibidas pela Figura 4.16 demonstram como os cenários de persistência em memória adicionam uma quantidade expressiva de processamento conforme mais falhas são induzidas. O cenário CH, por outro lado, tem sobrecargas de CF menores.

Assim, a diferença de desempenho entre os CFs indica que o custo de reprocessamento é maior que o custo de leitura. Como a leitura de partições perdidas não requisita a leitura do *dataset* por completo, a política CH conseguiu oferecer os menores níveis de sobrecarga nesse quesito. Apesar de não oferecer o melhor tempo de execução em nenhum dos testes, o salvamento em *checkpoint* mostrou-se conveniente para uma recuperação mais rápida.

A diferença de desempenho da política CH em relação a política MO, porém, dificulta a consolidação do cenário de *checkpoint* estático nos testes realizados. Portanto, percebe-se que a alternativa de *checkpoint* do *dataset* inicial do K-Means pode ser favorável apenas para cenários em que a confiabilidade do sistema é extremamente baixa. Embora exista um acréscimo no tempo de execução, uma grande quantidade de falhas pode compensar as sobrecargas geradas.

Todavia, a Figura 4.16 mostra que a diferença entre as sobrecargas de CF diminui conforme mais falhas são observadas. Com um menor número de *workers* disponíveis, a tendência do desempenho da política CH é ser ainda mais prejudicada. Logo, a queda de desempenho de execução do CH em momentos livres de falha passa a ser mais impactante para o desempenho da aplicação do que a eficiência de recuperação.

Sendo assim, o uso de *checkpoints* no K-Means deve ser planejado com cuidado, já que a política CH mostrou-se impeditiva em todos os casos testados, apesar da eficiência de recuperação. Esse comportamento impeditivo apresenta-se principalmente em cenários com mais iterações. Já as políticas MO e MAD mostraram um resultado satisfatório em cenários sem falhas, mas seus desempenhos de recuperação em CFs as tornam impeditivas em ambientes de baixa confiabilidade. Dessa maneira, o controle de persistência estático para o K-Means pode se tornar um grande desafio conforme o conhecimento sobre a plataforma de execução é limitado.

4.3.2.2 Políticas Dinâmicas

Nos testes com políticas dinâmicas, o K-Means foi executado com base na política MO, em que o RDD $data$ é marcado para persistência em memória. O restante dos *datasets*, assim como na abordagem estática, não foram modificados em virtude da limitação de código apresentada. Contudo, as políticas dinâmicas possuem a vantagem de observar *datasets* dentro da própria execução de uma aplicação. Desta forma, todos os *datasets* da Tabela 4.9 podem ser considerados para *checkpoint* através da DCA, de acordo com a política usada.

Dessa forma, a DCA foi testada no K-Means através das políticas FA (*Failure Awareness*) e LT (*Lineage Threshold*). Foram definidos dois *thresholds* para a política LT de 3 e 10 dependências. A escolha de 3 dependências se dá pela oportunidade de testar um cenário próximo ao estabelecimento de *checkpoint* para todos os *datasets*. Com 10 dependências, cria-se a oportunidade de verificar o comportamento do *checkpoint* apenas de *datasets* pertencentes a iterações de processamento (excluindo-se a entrada e o treinamento).

O K-Means com políticas dinâmicas foi executado através de 5, 10 e 15 iterações, além dos 3 CFs definidos para o *benchmark*. Os resultados de tempo de execução e desvio padrão obtidos são sumarizadas pela Tabela 4.13, para o cenário de carga com 500 milhões de pontos, e pela Tabela 4.14 para o cenário com 1 bilhão de pontos.

Política	**Cenário de Falha**	**Iterações**					
		5		**10**		**15**	
		Execução (s)	**Dv. Padrão (s)**	**Execução (s)**	**Dv. Padrão (s)**	**Execução (s)**	**Dv. Padrão (s)**
LT_3	S/Falha	1618,1	154,8	2086,6	100,2	2747,0	91,6
	CF_1	2121,4	161,4	2505,1	155,8	3004,6	137,0
	CF_2	3498,2	160,5	3741,0	151,6	4222,6	177,0
	CF_3	4593,0	178,4	5461,0	111,0	6210,1	163,7
LT_{10}	S/Falha	694,4	53,6	815,6	87,6	947,2	71,2
	CF_1	711,33	144,2	862,5	103,7	990,6	100,9
	CF_2	744,7	110,0	904,8	114,6	1001,8	102,6
	CF_3	978,0	125,6	1047,0	122,5	1105,9	131,7
FA	S/Falha	158,3	21,0	179,4	23,5	189,4	18,6
	CF_1	235,0	85,8	298,1	118,7	306,1	98,4
	CF_2	362,9	114,3	413,2	154,1	446,9	116,7
	CF_3	453,5	119,8	489,5	173,4	465,8	97,1

Tabela 4.13 – Resultados obtidos pelo K-Means com políticas dinâmicas e 500 milhões de pontos no Spark.

O tempo de execução obtido pelas políticas de *threshold* de *lineage* LT_3 e LT_{10} esclarecem a sensibilidade do K-Means quanto a política de persistência via *checkpoint*. Com um *lineage* pequeno, muitos *checkpoints* são realizados – inclusive $data$. Por consequência, o desempenho é semelhante ao apresentado pela política estática CH, com tempos de execu-

Política	Cenário de Falha	Iterações					
		5		10		15	
		Execução (s)	Dv. Padrão (s)	Execução (s)	Dv. Padrão (s)	Execução (s)	Dv. Padrão (s)
	S/Falha	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$
LT_3	CF_1	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$
	CF_2	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$
	CF_3	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$	$undefined$
	S/Falha	1899,5	99,4	2322,9	114,6	2640,3	125,3
LT_{10}	CF_1	2155,9	129,2	2787,0	156,2	3188,0	150,6
	CF_2	2590,1	146,8	3253,5	121,4	3698,0	179,6
	CF_3	2973,2	67,8	3684,7	119,2	4001,2	134,1
	S/Falha	1217,6	51,4	1498,7	53,5	1655,1	60,2
FA	CF_1	1346,8	56,7	1722,5	95,3	1845,6	68,1
	CF_2	1554,7	108,1	1955,1	90,7	2068,3	112,5
	CF_3	1993,5	104,5	2295,7	98,8	2532,7	121,8

Tabela 4.14 – Resultados obtidos pelo K-Means com políticas dinâmicas e 1 bilhão de pontos no Spark.

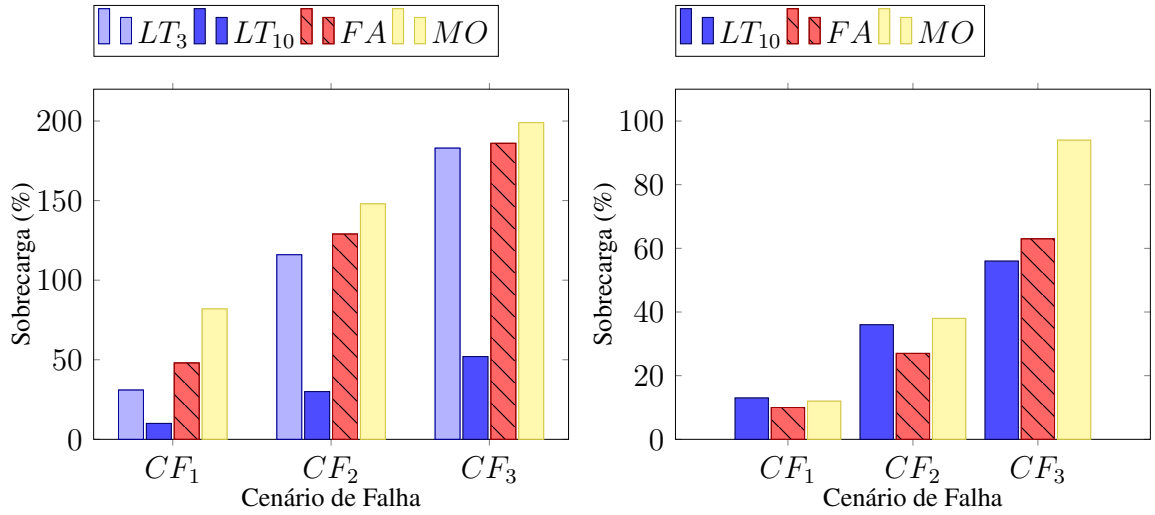
ção impeditivos. Inclusive, com 3 dependências as execuções também não foram completadas ($undefined$) no cenário de 1 bilhão de pontos, impossibilitando uma análise sobre a política nesse caso.

Quando o limiar é aumentado para 10 dependências, o desempenho da política possui uma grande alteração. O tempo de execução diminui consideravelmente, uma vez que o *dataset* $data$ deixa de ser considerado. Com 1 bilhão de pontos, uma análise dos resultados passa a ser possível. Ainda assim, a política LT_{10} realiza salvamentos de *checkpoint* em cenários sem falha, que justificam o tempo excedent em relação aos melhores casos do cenário estático.

Conforme esperado, a política FA ofereceu o melhor tempo de execução em cenários sem falha dentre as políticas dinâmicas testadas. Como o fator MTBF bloqueia o início da etapa 2 da política, nenhum *checkpoint* foi realizado. Por isso, a política obteve resultados satisfatórios, já que o cenário sem falha não é beneficiado por salvamentos de *checkpoint* – conforme já evidenciado por testes anteriores.

Já em cenários de falha, a política FA retornou os melhores resultados dentre todas as políticas testadas (estáticas e dinâmicas). Dessa forma, foi possível consolidar o dinamismo de *checkpoints* como uma solução eficiente tanto em cenários sem falha quanto com falha. A Figura 4.17 mostra como a sobrecarga de CF obtida pela política FA manteve-se abaixo das políticas dinâmicas LT.

Para fins de comparação, a Figura 4.17 também exibe a sobrecarga de CF da política MO. A baixa sobrecarga de CF apresentada por ambas as políticas dinâmicas revela o fato de que reprocessamentos são mais custosos que a leitura de *checkpoints* em grande parte da apli-



(a) Cenário com 500 milhões de pontos. (b) Cenário com 1 bilhão de pontos.

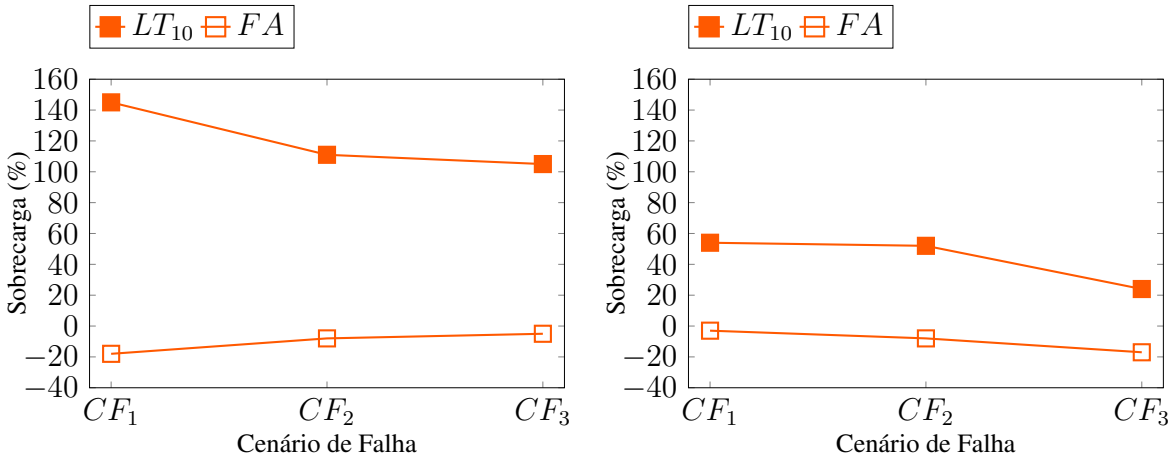
Figura 4.17 – Sobrecargas de CF das políticas dinâmicas no K-Means em 5 iterações com 500 milhões e 1 bilhão de pontos.

cação. Ainda que RDDs específicos – como o $data$ – mostrem uma característica contrária, os demais *datasets* da execução possuem uma complexidade de processamento alta que justificam a quebra do *lineage* e consequente recuperação via leitura do HDFS.

A diferença entre FA e MO se dá principalmente em cenários com maior quantidade de falhas (CF_3), dado que esse CF implica em uma maior quantidade de trabalho perdido no momento da falha. Como a política FA tratou salvamentos entre iterações, houve um caminho menor para o procedimento de recuperação percorrer. Isto é, o trabalho perdido na i-ésima iteração precisou recuperar dados de i iterações perdidas em cenários sem *checkpoint*. Com um *checkpoint* salvo na n-ésima iteração, a recuperação reprocessa $i - n$ iterações. A Figura 4.18 exibe a sobrecarga das políticas LT_{10} e FA sobre a política MO em cenários de falha.

A política FA apresentou ganhos de desempenho em relação a política MO em praticamente todos os cemários de execução com falhas. Por sua vez, a política LT_{10} apresentou altos níveis de sobrecarga com CF_1 que foram diminuídos conforme o número de falhas aumentou. A falta de uma observação sobre custos condicionou a política a salvar *datasets* inapropriados em *checkpoint*, o que tornou a sobrecarga sobre o MO alta apesar da baixa sobrecarga de CF.

Ainda assim, os *checkpoints* salvos pelas políticas dinâmicas durante as iterações não realizam uma quantidade grande de operações de escrita no HDFS, em sua maioria. Ainda que a quantidade de pontos seja grande o suficiente para gerar intrusividade quando o *dataset* completo é salvo em *checkpoint* (vide resultados do CH), os *datasets* intermediários para cálculo de distâncias entre as iterações possuem dados resumidos.



(a) Cenário com 500 milhões de pontos.

(b) Cenário com 1 bilhão de pontos.

Figura 4.18 – Sobrecargas das políticas dinâmicas no KMeans com 500 milhões e 1 bilhão de pontos em relação a política estática MO.

Consequentemente, o tamanho desses *datasets* é menor. As Tabelas 4.15 (500 milhões de pontos) e 4.16 (1 bilhão de pontos) auxilia na compreensão dessa conclusão com a quantidade e o custo médio de salvamento dos *checkpoints* estabelecidos pelas políticas FA e LT_{10} em CFs, além do percentual que esse tempo representa no tempo de execução total do respectivo cenário testado.

Política	**Cenário de Falha**	**T. Salvamento (s)**	**Sobrecarga**	**Qtd. Salvamentos**
LT_3	S/Falha	160,2	9,9%	9
	CF_1	148,9	7,0%	9
	CF_2	160,5	4,6%	9
	CF_2	188,8	4,1%	9
LT_{10}	S/Falha	36,1	5,6%	6
	CF_1	35,1	5,0%	6
	CF_2	34,3	4,1%	6
	CF_2	33,1	3,3%	6
FA	S/Falha	-	-	0
	CF_1	7,7	3,3%	7
	CF_2	1,7	0,5%	6,1
	CF_2	1,7	0,4%	4,5

Tabela 4.15 – Média da quantidade e do tempo de salvamento de *checkpoints* das políticas dinâmicas no K-Means com 5 iterações e 500 milhões de pontos.

Os custos de *checkpoint* da política LT_{10} são maiores que os custos da política FA, conforme as Tabelas 4.15 e 4.16. O número de *checkpoints* com um *threshold* de 10 dependências se mantém o mesmo, independentemente do número de iterações. Como o estabelecimento de *checkpoint* implica no corte de *lineage*, os novos RDDs possuem um número menor de depen-

Política dinâmica	Cenário de Falha	T. Salvamento (s)	Sobrecarga	Qtd. Salvamentos
LT_{10}	S/Falha	104,9	5,5%	6
	CF_1	129,1	6,0%	6
	CF_2	154,6	5,9%	6
	CF_2	122,6	4,1%	6
FA	S/Falha	-	-	0
	CF_1	1,4	0,1%	5,8
	CF_2	1,7	0,1%	5,9
	CF_2	1,8	0,1%	5,1

Tabela 4.16 – Média da quantidade e do tempo de salvamento de *checkpoints* das políticas dinâmicas no K-Means com 5 iterações e 1 bilhão de pontos.

dências. Porém, essa política também mostrou sobrecargas de CF menores em relação a política estática MO. Portanto, os resultados obtidos sumarizam o desempenho da política LT_{10} com uma abordagem eficiente em recuperação de falhas, mas intrusiva em cenários sem falha.

4.3.2.3 Discussão

Nos testes realizados com o *benchmark* K-Means, o desempenho apresentado pela técnica de *checkpoint* dependeu diretamente do *dataset* escolhido. Esse comportamento oferece insegurança para a aplicação, sendo que o tempo de execução pode ser impeditivo de acordo com escolhas inapropriadas. A política MO se mostrou extremamente eficiente em cenários de falha, mas seu desempenho de recuperação pós-falha evidenciou a necessidade do uso de *checkpoints* de forma controlada.

O uso de políticas dinâmicas no K-Means auxiliou na busca por cenários eficientes de *checkpoint*. A política LT_{10} mostrou que o estabelecimento de *checkpoints* serviu para a uma recuperação mais eficiente que o reprocessamento. Mas a dependência de um atributo estático não forneceu de forma total o comportamento dinâmico pretendido pela DCA. O cenário LT_3 foi importante para que o problema do *threshold* estático fosse melhor observado, já que seu desempenho foi similar ao da política CH.

Já a política FA destacou-se pelo desempenho alcançado nos cenários com e sem falha. Isto é, essa política foi a única dentre as abordagens dinâmicas e estáticas utilizadas que conseguiu fornecer um equilíbrio entre desempenho e confiabilidade de acordo com os cenários de teste induzidos. Portanto, a conclusão dos testes da DCA no K-Means mostra que a complexidade do algoritmo requer grande cuidado para a análise de desempenho.

Através dos resultados obtidos, também pode-se observar que a grande diferença entre

o bom desempenho da política FA para as políticas com *threshold* de *lineage* nos cenários de falha justificou-se pelo mesmo motivo dos cenários sem falha. A quantidade desnecessária e ineficiente (sem noção de custos) de salvamentos com a política LT comprometeu seu desempenho, principalmente em cenários sem falha em que o estabelecimento de *checkpoints* é descartável.

Por fim, detaca-se que as soluções de *checkpoint* dinamicamente configurado nas execuções do K-Means conseguiu explorar a persistência de *datasets* inacessíveis via código pelo HiBench. Essa característica mostrou-se essencial nos cenários de testes explorados, de modo que houve uma grande variação de desempenho e de confiabilidade conforme diferentes políticas foram usadas. Essa variação apresentou resultados menos eficientes, conforme os casos CH e LT_3 em cenários de falha. Mas a presença de *checkpoints* auxiliou em recuperações mais rápidas, o que torna o uso de *checkpoints* essencial em sistemas e plataforma de baixa confiabilidade.

5 CONSIDERAÇÕES FINAIS

A técnica de *Checkpoint and Recovery* é amplamente usada em sistemas computacionais de alto desempenho como forma de aumentar seus níveis de confiabilidade e disponibilidade. Contudo, configurações estáticas promovem uma queda na eficiência da técnica, à medida que diferentes cenários de execução são observados. Nesse sentido, o presente trabalho definiu a Dynamic Configuration Architecture (DCA) como proposta para conter as limitações de configurações estáticas nas implementações do *checkpoint* em dois *framework* de processamento distribuído: Apache Hadoop e Apache Spark.

A partir do uso da DCA, foi possível monitorar elementos do sistema e controlar a configuração da técnica de *checkpoint* em ambos os *frameworks*. O uso do histório se mostrou essencial para a estimativa de valores desconhecidos, sem a necessidade de deduções ou de observações intuitivas. Com estimativas baseadas nas experiências em tempo real fornecidas pelas execuções dos testes, foi possível definir custos e fatores de elementos que interferem diretamente no desempenho da técnica, como o tempo médio entre falhas (MTBF) e o custo computacional associado ao estabelecimento de *checkpoints*.

Ainda assim, a arquitetura de histórico do *coordinator* pode ser aprimorada através de análises mais rebuscadas. Neste trabalho, o histórico foi validado a partir de análises com médias aritméticas simples. Porém, cálculos mais apurados também podem ser usados. Uma abordagem de média ponderada pode ser usada para valorizar observações mais recentes. Para uma análise mais apurada, pode-se aplicar uma solução baseada em modelos de regressão. Em novas validações, esses aspectos deverão ser considerados.

Os testes de desempenho realizados ressaltaram, de forma geral, que o uso da DCA promoveu com sucesso uma adaptação em tempo real das implementações dos *frameworks* às observações do monitoramento. No Hadoop, os resultados obtidos com períodos dinamicamente configurados, baseados em aproximações de período ideal de *checkpoint*, alcançaram resultados eficientes. Ainda que a definição manual de um período ideal seja possível, a mudança de comportamento das aplicações e do sistema requer uma adaptação em tempo real desse atributo para que a eficiência da técnica de *Checkpoint and Recovery* mantenha-se presente.

As aproximações de Young e Daly, usadas para definir períodos ideais, atingiram resultados satisfatórios. Em cenários sem falha, a intrusividade de salvamentos em excesso foi controlada, ao contrário do período estático frequente testado (10 segundos). Ademais, a quan-

tidade de *checkpoints* alocados em momentos estratégicos das execuções explica o baixo tempo de execução em cenários de falha no elemento mestre do HDFS. Isto é, as aproximações usadas auxiliaram na busca pelo cenário ideal em que o tempo gasto para a recuperação de uma falha é pequeno. Sendo assim, o equilíbrio proposto pela DCA foi alcançado com sucesso.

Nesse sentido, a aplicação de outras aproximações de período entre *checkpoints* no Hadoop deverá ser avaliada em trabalhos futuros. Assim, pode-se verificar o comportamento da técnica de *checkpoint* no Hadoop através de uma maior variedade de fatores, identificando melhores casos para aplicações específicas.

No Spark, a DCA foi validada com variações nas políticas de *checkpoint* dinâmico. Para isso, foram usadas as políticas LT e FA em dois *benchmarks* de propósito semelhante, mas com características de código distintas. Os resultados obtidos nessas validações mostraram que o uso de *checkpoints* pode ser essencial em aplicações como o PageRank, em que um reuso de dados não é feito efetivamente mesmo se tratando de uma aplicação iterativa.

Por outro lado, aplicações com um alto reuso de dados mostraram que o *checkpoint* pode ser extremamente prejudicial quando submetido de forma negligente. Esse foi o caso do K-Means, em que o estabelecimento do *checkpoint* no *dataset* de entrada tornou impraticável a execução do *benchmark*. Ainda assim, percebeu-se também que o uso de *checkpoint* em outras regiões do código garantiu um melhor tempo de recuperação em cenários de falha.

Nesses *benchmarks*, a DCA provou que decisões sobre *checkpoints* em aplicações no Spark podem ser úteis em caso de recuperação, mas geram intrusividade em cenários livres de falha. Desta forma, a política FA apresentou os melhores resultados ao fugir de estabelecimentos estáticos de *threshold*, adaptando-se com sucesso ao contexto de execução. Ao final, percebe-se que a política obteve desempenho idêntico à política MO sem falhas e próximo às políticas de *checkpoint* com tempos de recuperação eficientes quando falhas foram observadas.

A contribuição da DCA em cenários de falha do Spark – especificamente no caso do K-Means – está diretamente ligada à possibilidade de estabelecimento de *checkpoint* em *datasets* inacessíveis via código. Nos casos em que o acesso ao código fonte de aplicações (ou parte de aplicações) é limitado, a alternativa de *checkpoints* dinâmicos com a DCA desempenha uma observação interna da aplicação que é inviável na abordagem padrão de estabelecimento de *checkpoints* do Spark.

Portanto, observa-se que a DCA alcançou resultados satisfatórios nos testes com ambos os *frameworks* Hadoop e Spark. Porém, os resultados podem ser aprimorados conforme as mé-

tricas e as políticas de monitoramento são otimizadas. Através dos trabalhos futuros propostos, uma validação mais completa deverá ser realizada. No Hadoop, novas aproximações para o cálculo de períodos ideais entre *checkpoints* serão estudadas.

No Spark o uso da política MT será explorada, de forma que as políticas de seleção definidas pela DCA também sejam analisadas em execução. Além disso, a otimização da política FA deve ser trabalhada em novos trabalhos. Com uma melhor adaptação aos custos e a análise de falhas, a política pode alcançar resultados ainda melhores do que os apresentados neste trabalho.

Por fim, a possibilidade de expandir as funcionalidades da DCA para outros *frameworks*, sistemas e/ou técnicas deve ser destacada. Como o foco da arquitetura é a configuração dinâmica de atributos, entende-se que o aproveitamento desse conceito em outros contextos pode gerar resultados satisfatórios em diversos casos.

REFERÊNCIAS

ACETO, G. et al. Cloud monitoring: a survey. **Computer Networks**, [S.l.], v.57, n.9, p.2093–2115, 2013.

ALTMAN, A.; TENNENHOLTZ, M. Ranking systems: the pagerank axioms. In: ACM CONFERENCE ON ELECTRONIC COMMERCE, 6. **Proceedings...** [S.l.: s.n.], 2005. p.1–8.

ARTHUR, D.; VASSILVITSKII, S. k-means++: the advantages of careful seeding. In: ACM-SIAM SYMPOSIUM ON DISCRETE ALGORITHMS. **Proceedings...** [S.l.: s.n.], 2007. p.1027–1035.

BAHMANI, B. et al. Scalable k-means++. **Proceedings of the VLDB Endowment**, [S.l.], v.5, n.7, p.622–633, 2012.

BALOUEK, D. et al. Adding Virtualization Capabilities to the Grid'5000 Testbed. In: IVANOV, I. et al. (Ed.). **Cloud Computing and Services Science**. [S.l.]: Springer International Publishing, 2013. p.3–20. (Communications in Computer and Information Science, v.367).

BARTLETT, W.; SPAINHOWER, L. Commercial fault tolerance: a tale of two systems. **IEEE Transactions on dependable and secure computing**, [S.l.], v.1, n.1, p.87–96, 2004.

CARDOSO, P. V.; BARCELOS, P. P. Validation of a dynamic checkpoint mechanism for Apache Hadoop with failure scenarios. In: TEST SYMPOSIUM (LATS), 2018 IEEE 19TH LATIN-AMERICAN. **Anais...** [S.l.: s.n.], 2018. p.1–6.

CARDOSO, P. V.; BARCELOS, P. P. Performance Evaluation of Apache Hadoop Benchmarks under a Dynamic Checkpointing Mechanism. In: INTERNATIONAL CONFERENCE OF THE CHILEAN COMPUTER SCIENCE SOCIETY (SCCC), 2018. **Anais...** [S.l.: s.n.], 2018. p.1–7.

DALY, J. A model for predicting the optimum checkpoint interval for restart dumps. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL SCIENCE. **Anais...** [S.l.: s.n.], 2003. p.3–12.

DALY, J. T. A higher order estimate of the optimum checkpoint interval for restart dumps. **Future generation computer systems**, [S.l.], v.22, n.3, p.303–312, 2006.

DEAN, J.; GHEMAWAT, S. MapReduce: a flexible data processing tool. **Communications of the ACM**, [S.l.], v.53, n.1, p.72–77, 2010.

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.

EDUREKA. **Big Data and Hadoop Tutorial**. [S.l.: s.n.], 2019. http://www.slideshare.net/EdurekaIN/hadoop-week1-release22 (acessado em julho de 2019).

EGWUTUOHA, I. P. et al. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. **The Journal of Supercomputing**, [S.l.], v.65, n.3, p.1302–1326, 2013.

EL-SAYED, N.; SCHROEDER, B. Checkpoint/restart in practice: when 'simple is better'. In: CLUSTER COMPUTING (CLUSTER), 2014 IEEE INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2014. p.84–92.

FOUNDATION, A. S. **Apache Spark**: lightning-fast cluster computing. [S.l.: s.n.], 2019. https://spark.apache.org/ (acessado em julho de 2019).

FOUNDATION, A. S. **Apache Hadoop 2.7.3**. [S.l.: s.n.], 2019. https://hadoop.apache.org/docs/r2.7.3/ (acessado em julho de 2019).

GHIT, B.; EPEMA, D. Better Safe than Sorry: grappling with failures of in-memory data analytics frameworks. In: INTERNATIONAL SYMPOSIUM ON HIGH-PERFORMANCE PARALLEL AND DISTRIBUTED COMPUTING, 26. **Proceedings...** [S.l.: s.n.], 2017. p.105–116.

GOPALANI, S.; ARORA, R. Comparing apache spark and map reduce with performance analysis using k-means. **International journal of computer applications**, [S.l.], v.113, n.1, 2015.

HUANG, S. et al. The HiBench benchmark suite: characterization of the mapreduce-based data analysis. In: IEEE 26TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING WORKSHOPS (ICDEW 2010), 2010. **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, p.9.

INTERLANDI, M. et al. Titian: data provenance support in spark. **Proceedings of the VLDB Endowment**, [S.l.], v.9, n.3, p.216–227, 2015.

JAIN, H.; GOYAL, A. An Improved Approach for Analysis of Hadoop Data for All Files. **International Journal of Computer Applications**, [S.l.], v.157, n.4, 2017.

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.

LAPRIE, J.-C. Dependable computing and fault tolerance: concepts and terminology. In: TWENTY-FIFTH INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 1995,. **Anais...** [S.l.: s.n.], 1985. p.2.

LASKOWSKI, J. **Mastering Apache Spark 2**. [S.l.: s.n.], 2019. https://jaceklaskowski.gitbooks.io/mastering-apache-spark/ (acessado em julho de 2019).

LEVITIN, G. et al. Dynamic Checkpointing Policy in Heterogeneous Real-Time Standby Systems. **IEEE Transactions on Computers**, [S.l.], 2017.

LIU, H. et al. Performance and energy modeling for live migration of virtual machines. **Cluster computing**, [S.l.], v.16, n.2, p.249–264, 2013.

MASHAYEKHY, L. et al. Energy-aware scheduling of mapreduce jobs for big data applications. **IEEE transactions on Parallel and distributed systems**, [S.l.], v.26, n.10, p.2720–2733, 2014.

NAKSINEHABOON, N. et al. Reliability-aware approach: an incremental checkpoint/restart model in hpc environments. In: INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID (CCGRID, 2008. **Anais...** [S.l.: s.n.], 2008. p.783–788.

NETZER, R. H.; XU, J. Adaptive message logging for incremental program replay. **IEEE Parallel & Distributed Technology: Systems & Applications**, [S.l.], v.1, n.4, p.32–39, 1993.

PAGE, L. et al. **The PageRank citation ranking**: bringing order to the web. [S.l.]: Stanford InfoLab, 1999.

POLO, J. et al. Resource-aware adaptive scheduling for mapreduce clusters. In: INTERNATIONAL MIDDLEWARE CONFERENCE, 12. **Proceedings...** [S.l.: s.n.], 2011. p.180–199.

PULLUM, L. L. **Software fault tolerance techniques and implementation**. [S.l.]: Artech House, 2001.

ROMAN, E. A survey of checkpoint/restart implementations. In: LAWRENCE BERKELEY NATIONAL LABORATORY, TECH. **Anais...** [S.l.: s.n.], 2002.

SHAFER, J.; RIXNER, S.; COX, A. L. The hadoop distributed filesystem: balancing portability and performance. In: PERFORMANCE ANALYSIS OF SYSTEMS & SOFTWARE (ISPASS), 2010 IEEE INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2010. p.122–133.

SHI, J. et al. Clash of the titans: mapreduce vs. spark for large scale data analytics. **Proceedings of the VLDB Endowment**, [S.l.], v.8, n.13, p.2110–2121, 2015.

TANENBAUM, A. S.; VAN STEEN, M. **Distributed systems**: principles and paradigms. [S.l.]: Prentice-Hall, 2007.

VERMA, J. P.; PATEL, A. Comparison of MapReduce and Spark Programming Frameworks for Big Data Analytics on HDFS. **International Journal of Computer Science and Communication**, [S.l.], v.7, n.2, p.80–84, 2016.

VISWANATH, V. **Spark RDDs Simplified**. [S.l.: s.n.], 2016. http://vishnuviswanath.com/spark_rdd.html (acessado em julho de 2019).

WHITE, T. **Hadoop**: the definitive guide, 4th edition. [S.l.]: "O'Reilly Media, Inc.", 2015.

YOUNG, J. W. A first order approximation to the optimum checkpoint interval. **Communications of the ACM**, [S.l.], v.17, n.9, p.530–531, 1974.

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. p.2–2.