Validação de Políticas para o Estabelecimento Dinâmico de Checkpoints no Apache Spark

Paulo V. M. Cardoso, Rhauani Weber Aita Fazul, Patrícia Pitthan Barcelos

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

{pcardoso, rwfazul, pitthan}@inf.ufsm.br

Abstract. *Apache Spark is a platform designed for in-memory distributed data processing. For a reliable and fault-tolerant persistence, it uses the checkpointing technique. Establishing checkpoints on Spark, however, needs to be done manually in the source code, which makes efficient setup a big challenge. This paper presents and validates a dynamic configuration architecture for checkpoints in Spark. The proposed architecture initiates checkpoint procedures automatically, based on monitoring policies that observe the system and the applications. The evaluation results show that using suitable dynamic policies can increase Spark's reliability without compromising its performance.*

Resumo. *O Apache Spark é uma plataforma voltada ao processamento distribuído de dados em memória. Para uma persistência confiável e tolerante a falhas, o Spark usa a técnica de checkpoint. O estabelecimento de checkpoints, entretanto, precisa ser realizado manualmente através do código-fonte, o que dificulta uma configuração eficiente. Esse trabalho apresenta e valida uma arquitetura de configuração dinâmica para checkpoints no Spark. A arquitetura proposta inicia procedimentos de checkpoint automaticamente, com base em políticas de monitoramento que observam o sistema e suas aplicações. Os experimentos demostram que o uso de políticas dinâmicas adequadas é capaz de aumentar a confiabilidade do Spark sem comprometer seu desempenho.*

1. Introdução

Conforme a produção de dados digitais cresce, a demanda por sistemas computacionais de alto desempenho exige maior eficiência. Uma grande variedade de arquiteturas de alto desempenho é utilizada para esse propósito. Contudo, fatores como confiabilidade e disponibilidade de um sistema são prejudicados à medida que sua arquitetura torna-se mais complexa. Por isso, a aplicação de técnicas de tolerância a falhas (TF) é essencial para evitar erros computacionais decorrentes de falhas. Uma técnica de TF bastante utilizada é a recuperação de erros, a qual pode ocorrer por avanço ou por retorno [Laprie 1985].

O Apache Spark [Foundation 2019] é um *framework* para processamento distribuído, que faz uso de *checkpoints* como uma técnica de recuperação de erros por retorno. No Spark, existe a possibilidade de armazenamento de dados em memória. Como a quantidade de memória disponível em sistemas computacionais é escassa quando comparada ao disco, faz-se necessário um gerenciamento de persistência eficiente. Quando

o espaço disponível não atende à demanda, o disco vira um elemento chave para a continuidade da aplicação. Logo, o *checkpoint* apresenta-se como uma técnica útil para o armazenamento de conteúdo em disco.

Contudo, esta técnica possui uma grande limitação no Spark, onde o estabelecimento de *checkpoints* deve ser definido pelo desenvolvedor da aplicação via código. Essa característica dificulta a escolha pelo melhor momento para salvar dados em *checkpoint*, uma vez que o desenvolvedor deve ter um conhecimento detalhado da aplicação e do comportamento do Spark de modo a não criar gargalos de desempenho. Além disso, é preciso ter um conhecimento acerca de quais dados devem ser enviados para o disco.

O uso mais eficiente do *checkpoint* no Spark tem sido estudado por outros autores. No trabalho de [Zhu et al. 2016], o funcionamento manual de *checkpoints* e a intervenção do desenvolvedor também foram notados. Como solução, o trabalho define um mecanismo de salvamento automático para os *checkpoints* de acordo com a relevância dos dados e o uso de memória, a fim de eliminar futuras sobrecargas de recomputação. Contudo, a solução apresentada não considera os custos envolvidos no procedimento de *checkpoint*. Já o trabalho proposto em [Yan et al. 2016] apresenta uma solução de *checkpoint* baseada na complexidade dos dados e na probabilidade de falhas. A solução, porém, exige que a distribuição de falhas e o custo de processamento sejam conhecidos *a priori*.

Para contornar os desafios apresentados, abordagens voltadas à configuração dinâmica de *checkpoints* podem ser utilizadas. Uma solução para tal configuração é a *DCA: Dynamic Configuration Architecture*. Esta arquitetura já foi previamente validada [Cardoso and Barcelos 2018b] e apresentou resultados eficientes em relação à adaptação do período entre *checkpoints* no Apache Hadoop: um *framework open source* amplamente utilizado para processamento e armazenamento eficiente de *big data* [White 2015].

Neste trabalho apresentamos uma possibilidade diferente para a DCA visando uma adaptação dinâmica da técnica de *checkpoint* implantada no *framework* Apache Spark. Para isso, adaptamos a DCA para trabalhar com dois propósitos principais: a observação do comportamento do sistema em tempo de execução; e a autonomia do Spark na tomada de decisões a respeito do procedimento de *checkpoint*. O objetivo da arquitetura no Spark é passar a responsabilidade das políticas de persistência – anteriormente dos desenvolvedores – para o próprio *framework*.

As decisões sobre quais dados são salvos em *checkpoint* no Spark pela DCA partem de políticas de monitoramento. Essas políticas definem quando um procedimento de *checkpoint* deve ser iniciado no Spark, de modo a suprir alguma limitação observada. Neste trabalho, apresentamos as políticas: (i) *Lineage Threshold* (LT), que define um limiar máximo para o tamanho de uma *lineage*; e (ii) *Failure Awareness* (FA), que calcula o custo computacional de um *dataset* aliado à probabilidade de falhas no sistema.

Para validar a adaptação da DCA no Spark, bem como as políticas de monitoramento implementadas, experimentos foram realizados através de testes de desempenho pelo *benchmark* K-means [Meng et al. 2016]: uma aplicação CPU *bound* iterativa com opção de processamento distribuído pelo Spark. Diferentes experimentos com indução de falhas foram idealizados de modo a simular um ambiente de baixa confiabilidade.

O trabalho está organizado em 5 seções. A Seção 2 descreve o *framework* Apache Spark. A Seção 3 apresenta a arquitetura de configuração dinâmica para *checkpoint*, além

de detalhar o seu funcionamento, sua implementação no Spark e as políticas de monitoramento definidas. A Seção 4 discute os experimentos e resultados obtidos. Por fim, a Seção 5 apresenta as considerações finais e direciona os trabalhos futuros.

2. Apache Spark

O Apache Spark é um *framework open source* que oferece uma plataforma voltada para computação distribuída [Karau and Warren 2017]. O uso de processamento em memória é uma das grandes vantagens do Spark, proporcionando rapidez em operações de escrita e leitura de dados [Verma and Patel 2016]. A arquitetura do Spark conta com a definição de um único mestre (*master*) que gerencia um ou mais trabalhadores (*workers*).

Uma aplicação Spark é representada pelo componente $SparkContext$, e suas requisições são atendidas por *executors*: elementos que rodam em *workers* a fim de processar as operações e tarefas submetidas. Em geral, os dados em uma aplicação são divididos em diversas partições, que são armazenadas e processadas por diferentes nós de um *cluster* através dos *executors*.

Para gerenciar dados em memória, o Spark se aproveita de uma abstração de dados chamada *Resilient Distributed Dataset* (RDD) [Foundation 2019]. RDDs são *datasets* que podem ser divididos em partições (*partitions*) e distribuídos através do *cluster*. Além da definição dos dados, um RDD armazena a sua relação de dependência com outros *datasets*, formando uma linha do tempo (*lineage*) capaz de garantir recuperações em casos de falha de alguma partição a partir da sua reconstrução.

Existem duas maneiras de trabalhar com RDDs: a partir de transformações ou de ações. Transformações são operações que retornam um novo RDD, de forma a submeter uma alteração no *dataset*. Já as ações demandam um valor final ou geram uma saída (*output*) a um repositório externo [Karau and Warren 2017]. Um importante aspecto das operações em RDDs é o processamento baseado no modelo *lazy operation*: um *dataset* não é computado no mesmo instante em que é definido, mas apenas quando um resultado final (ação) é requisitado. O processamento de uma ação é definido como um *job*, em que todas as transformações são executadas para se chegar ao resultado final.

Em caso de novas ações sobre o mesmo RDD, a computação é realizada novamente. Para evitar essa repetição, pode-se persistir o conteúdo que já foi processado, para reuso em novas operações sobre o mesmo *dataset*. O Spark oferece opções de persistência em: (i) memória ($memory - only$); (ii) disco ($disk - only$); (iii) memória e disco ($memory - and - disk$); e (iv) *checkpoint*.

O *checkpoint* consiste no armazenamento de *datasets* em um repositório seguro. Essa opção difere-se das demais pela eliminação total da *lineage*. Ou seja, o *checkpoint* trunca a *lineage* de determinado RDD antes de persisti-lo, tornando-o um RDD raiz. Esse procedimento pode auxiliar em casos de recuperação pós-falha.

3. Dynamic Configuration Architecture (DCA)

O *checkpoint* apresenta-se como uma importante técnica no contexto de TF. Porém, configurá-lo de forma eficiente é um grande desafio, uma vez que diversos fatores podem interferir no seu comportamento. Uma alternativa já proposta é o uso de uma arquitetura para configuração dinâmica para atributos de *checkpoint*, denominada *Dynamic Configuration Architecture* (DCA) [Cardoso and Barcelos 2018b].

Inicialmente projetada para suporte ao *checkpoint* do Apache Hadoop, a DCA também foi proposta para outros *frameworks* [Cardoso and Barcelos 2018a]. Neste trabalho, aplicamos diferentes estratégias e políticas para adaptar a DCA para o Apache Spark. A implementação no Spark possui novos desafios. Por padrão, a criação de *checkpoints* nesse *framework* é definida via código pelo desenvolvedor: um processo custoso e propenso a escolhas ineficientes.

Com atributos adaptados dinamicamente, espera-se que a técnica de *checkpoint* tenha suas propriedades de TF melhor exploradas, ao mesmo tempo em que sua intrusividade seja controlada a partir da observação do sistema. Assim, o objetivo da DCA é tornar a configuração de atributos relacionados ao *checkpoint* mais eficiente em *frameworks* como o Spark. Para isso, um componente *monitor* e um *coordenador* de atributos são definidos, conforme mostra a Figura 1.

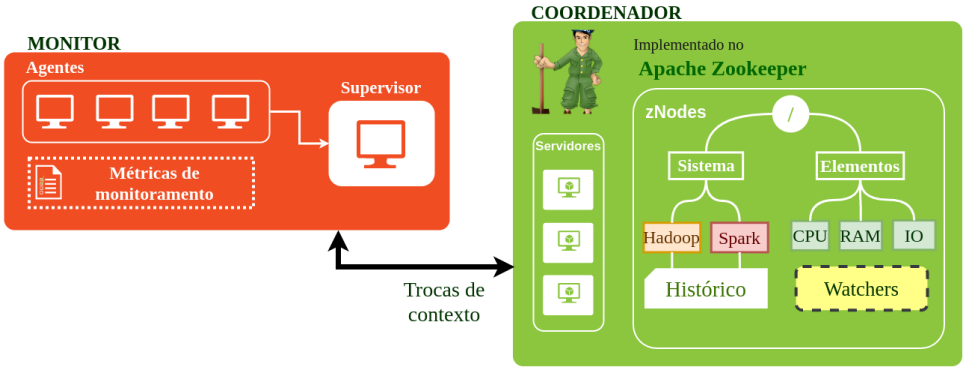

Figura 1. Arquitetura geral da DCA [Cardoso and Barcelos 2018a].

O *monitor* envia alertas para o *coordenador* sobre mudanças de contexto observadas no sistema. Uma mudança de contexto indica que o comportamento de um elemento do sistema sofreu alguma alteração desde o último monitoramento. Já o *coordenador* é implementado para gerar alertas às implementações nos *frameworks* (Hadoop ou Spark), além de manter uma comunicação em dois sentidos com o *monitor*. O *coordenador* também mantém os dados coletados no monitoramento em uma estrutura de árvore com o auxílio do *framework* Apache Zookeeper [White 2015]. Neste caso, cada recurso observado e cada sistema envolvido possuem seus próprios nós na árvore.

Quando há alteração na composição do *coordenador*, devido a mudanças de contexto recebidas do *monitor*, um mecanismo de alerta é executado. Qualquer implementação que esteja observando o *coordenador* também é notificada. Assim, o *coordenador* 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 ao *coordenador* para que recebam alertas sobre o andamento do sistema.

Um importante recurso do *coordenador* é o histórico de atributos, o qual mantém os dados coletados pelo *monitor*. Esses dados são armazenados em nós do Apache ZooKeeper, formando uma estrutura circular que funciona como uma janela de eventos. Assim, uma visão do comportamento do sistema no decorrer do tempo pode ser obtida, permitindo a definição – em tempo real – de fatores dificilmente conhecidos *a priori*, como o tempo médio entre falhas (*Mean Time Between Failures*, ou MTBF) e o tempo desde a última falha observada (*Time Since Last Failure*, ou TSLF) [Egwutuoha et al. 2013].

No Spark, a ativação da arquitetura segue o modelo dos atributos de configuração do *framework*. O atributo $spark.automatic.checkpoint$ deve ser setado no arquivo de configurações padrão. Contudo, o uso da DCA não exclui a possibilidade da criação de *checkpoints* via código, já que essa funcionalidade não foi modificada. Ou seja, o uso de *checkpoint* dinâmico não é tratado como um fator obrigatório mesmo com a arquitetura ativada. Pode-se, então, definir a funcionalidade da arquitetura como uma opção ao desenvolvedor Spark para uma tomada de decisões eficiente quanto a *checkpoints*.

O monitoramento da utilização de recursos é parte essencial da criação automática 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, essas políticas também foram definidas para escolher qual RDD deve ser salvo em *checkpoint*.

3.1. Políticas de Monitoramento

As políticas de monitoramento visam realizar escolhas automáticas e adaptadas ao contexto do sistema, relacionadas tanto a sua utilização geral, quanto às características dos *datasets* usados. Na prática, uma política de monitoramento define quando um procedimento de *checkpoint* deve ser iniciado no Spark. A principal vantagem da definição dessas métricas é a possibilidade de que *checkpoints* sejam estabelecidos 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 o desempenho de uma aplicação. As políticas de monitoramento usam monitores da DCA internos ao Spark com o intuito de identificar características de RDDs assim que eles são submetidos para armazenamento em memória e/ou processados. A função do *monitor* neste caso é identificar situações em que o procedimento de *checkpoint* é favorável, de acordo com as políticas estabelecidas. Neste trabalho, definimos duas políticas para o monitoramento de RDDs: o limite máximo de tamanho da *lineage* (*Lineage Threshold*, ou LT) e o custo computacional aliado à probabilidade de falhas (*Failure Awareness*, ou FA). Ressaltamos que ambas as políticas detalhadas a seguir fazem parte da DCA e, portanto, são contribuições originais baseadas no funcionamento dos RDDs.

3.1.1. Lineage Threshold (LT)

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

A política de monitoramento *Lineage Threshold* (LT) busca amenizar a complexidade de *jobs* com RDDs cuja *lineage* é maior que um limite preestabelecido. A partir do salvamento desses *datasets* em um *checkpoint*, suas *lineages* são truncadas e perdem informações de dependência. Esse cenário mostra-se atrativo quando o custo de leitura de um RDD salvo em um *checkpoint* é menor do que o custo de sua recomputação.

Nesta política, o monitoramento não é periódico e somente é iniciado quando 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 realizada se o tamanho de sua *lineage* for maior que um *threshold* definido nos arquivos de configuração. Caso uma perspectiva favorável para *checkpoint* – comandada pelo *threshold* escolhido pelo usuário com base em seu conhecimento do sistema e da aplicação – seja identificada, o RDD recebe o comando $rdd.checkpoint()$ logo após ser persistido.

O tamanho da *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 sua *lineage* calculada, 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 origem. 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 a reconstrução de um *dataset* é a principal justificativa de uso da política LT.

3.1.2. Failure Awareness (FA)

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 reprocessar os dados perdidos à medida que eventos de falha são observados nos nodos. A política de monitoramento *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 três etapas antes do RDD ser marcado para *checkpoint*. Na primeira etapa, dados referentes às falhas no sistema, sendo eles o tempo médio entre falhas (MTBF) e o tempo decorrido desde a última falha observada (TSLF), são obtidos do histórico de atributos implementado pela DCA [Cardoso and Barcelos 2018b]. Com estes dados, tem-se uma previsão do momento em que uma nova falha deve acontecer baseada em experiências anteriores.

Também utilizando o histórico, a política recupera o tempo previsto para a conclusão do *job* baseado na *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 de atributos.

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 1.

$$TE_{rdd} = jobTime * rdd.numPartitions \quad (1)$$

Com a estimativa do tempo de conclusão de um *job* e os fatores MTBF e TSLF, a primeira etapa da política é, então, executada. Esta etapa somente é concluída quando TE_{rdd} for maior que o tempo previsto para uma nova falha (que consiste em $MTBF - TSLF$), pois 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.

Após a validação da primeira etapa, a segunda etapa é executada 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 baixa complexidade sejam submetidos a *checkpoint*. O salvamento de *datasets* pouco complexos 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, uma terceira etapa é considerada: o tempo médio de leitura no repositório seguro usado para *checkpoints*. Neste trabalho, utilizamos o *Hadoop Distributed File System* (HDFS): um sistema de arquivos distribuído do ambiente Hadoop. O tempo médio de leitura no HDFS é obtido a partir da API de métricas disponibilizada pelo HDFS [White 2015]. Nesse caso, a DCA obtém o tempo médio de leitura de um bloco no HDFS ($ReadBlockAvgTime$). 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 2.

$$HRT_{rdd} = ReadBlockAvgTime * rdd.numPartitions \tag{2}$$

Logo, 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.

4. Experimentos

A fim de validar a proposta da DCA no Spark, foram realizados experimentos na plataforma Grid'5000[1]: um ambiente distribuído de larga escala com infraestrutura para testes relacionados a aplicações distribuídas e paralelas. Foi utilizado um *cluster* de 8 nodos, cada um com 2 CPUs Intel Xeon E5-2630 v3 (8-core), 128GB de RAM e 558GB disponível em disco. O Spark foi utilizado em sua versão 2.4.1.

A validação foi realizada na plataforma *HiBench*, através de uma interface de entrada para a implementação do algoritmo K-means (via biblioteca *MLlib* do Spark) [Meng et al. 2016]. O K-means é caracterizado como um *benchmark CPU intensive* de agrupamento iterativo que consiste no particionamento de valores em um ambiente n-dimensional em k *clusters* baseado na distância entre pontos [Foundation 2019]. Cada *cluster* é definido por um valor central presente no espaço de valores (centroide). Inicialmente, os k centroides são definidos de forma aleatória a partir de i etapas de inicialização.

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

4.1. Metodologia

De modo a validar os métodos de persistência já disponibilizados pelo Spark, foram definidas as políticas estáticas. Sendo assim, os testes avaliaram as políticas de persistência estáticas MO (*memory-only*), MAD (*memory-and-disk*) e CH (*checkpoint*). Para facilitar a análise das políticas de persistência, foram definidos RDDs alvo de acordo com o código do K-means. Esses RDDs consistem em todos os *datasets* do algoritmo que são passíveis de persistência estática ou dinâmica. Para políticas estáticas, os RDDs alvo foram marcados para persistência de acordo com a política definida. Sendo assim, a definição dos *datasets* alvo é essencial quando abordagens estáticas são usadas, devido à necessidade de conhecimento do usuário sobre o código da aplicação.

Na abordagem dinâmica, foram usadas as duas políticas de monitoramento implementadas na DCA: *Lineage Threshold* (LT) e *Failure Awareness* (FA). Nessa abordagem, os RDDs alvo não são necessários, já que as políticas de monitoramento decidem os RDDs a serem persistidos de forma automática. A política *threshold* de *lineage* foi testada com limiares definidos em 3 (LT_3) e 10 (LT_{10}) dependências. O caso LT_3 visa validar um cenário mais rígido, em que a maior parte dos RDDs deve ser considerada para *checkpoint*. Com 10 dependências, o intuito é validar situações cujos RDDs iniciais são descartados e somente RDDs com *lineages* mais longas são considerados.

Apesar dos RDDs alvo não serem necessários para as políticas da abordagem dinâmica, os experimentos realizados usam esses *datasets* como base para comparação com os resultados da abordagem estática. A Tabela 1 mostra os *datasets* alvo, além do método de persistência estática padrão usado pelo *HiBench*, para execuções com n iterações e i etapas de inicialização.

Tabela 1. RDDs alvo das execuções do K-means no Spark.

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

Dentre os *datasets* da Tabela 1, apenas o RDD $data$ pode ser manipulado via código do *HiBench*, de modo que o restante é encapsulado pela biblioteca *MLlib*. Por consequência, $data$ é o único *dataset* que foi modificado manualmente por políticas de persistência estáticas neste trabalho. O RDD $data$ corresponde ao conjunto de pontos do arquivo de entrada e é passado para o algoritmo assim que a *MLlib* é chamada. O RDD $norms$ armazena os dados normalizados. Já $costs$ define os pontos e suas distâncias aos centroides e é atualizado i vezes. Por fim, $centers$ surge para armazenar informações sobre novos centroides quando o processamento do algoritmo é efetivamente iniciado, de modo que esse *dataset* é modificado n vezes (sendo n iterações).

4.2. Cenários de Falha

A fim de criar a condição de falhas no Spark, foram definidos cenários de falha (CF). O objetivo foi analisar o comportamento das etapas de recuperação frente a eventos de falha 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 $kill$ do Linux foi usado para interromper o processo de um ou mais *workers*.

Os cenários de falha foram definidos com base 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 a política de persistência estática *memory-only* (MO) – sem falhas. 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*.

4.3. Resultados Obtidos

O arquivo de entrada das execuções foi gerado com 500 milhões de pontos, totalizando uma carga de 48GB de dados de entrada. Foram feitas validações com a geração de 10 *clusters* de valores com pontos de 3 dimensões. Cenários de teste foram construídos com base na carga de dados apresentada, no cenário de falha (CF), no número de iterações usadas no K-means (5, 10 e 15), nas políticas de persistência (para a abordagem estática) e nas políticas de monitoramento (para a abordagem dinâmica).

4.3.1. Políticas de Persistência Estáticas

Os resultados obtidos pelas políticas estáticas são sumarizados na Tabela 2, que exibe o tempo de execução e o desvio padrão dos diferentes cenários de teste. O desempenho das políticas MO e MAD mostrou-se praticamente idêntico, já que não houve sobrecarga de dados suficiente para que *datasets* tenham sido movidos da memória para o disco. Por outro lado, os testes com a política CH ressaltaram perda de desempenho causada pelo frequente acesso ao disco pelo *benchmark*. Os resultados apontam que o tempo de execução aumenta consideravelmente quando o *dataset* $data$ é persistido em *checkpoint*.

Tabela 2. Resultados obtidos pelas políticas de persistência estáticas do Spark.

Política	Cenário de Falha	Iterações					
		5		10		15	
		Execução (s)	Dv. Padrão (s)	Execução (s)	Dv. Padrão (s)	Execução (s)	Dv. Padrão (s)
MO	S/Falha	159,2	26,4	175,9	20,7	186,1	36,9
	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
MAD	S/Falha	158,9	20,1	177,6	18,4	188,8	28,1
	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
CH	S/Falha	1598,0	114,9	1999,0	122,7	2518,0	160,1
	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

De acordo com a Tabela 2, a política de persistência CH apresentou uma situação de intrusividade gerada por um baixo desempenho. O alto nível de intrusividade pode ser melhor entendido pelo custo de salvamento envolvido. A Tabela 3 exibe o tempo de salvamento do RDD $data$ em *checkpoint* nos diferentes cenários de falha, em execuções com 5 iterações, além do percentual que esse valor representa no tempo de execução.

Tabela 3. *Checkpoint* do *dataset* $data$ com a política estática CH (5 iterações).

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%

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 partições. 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.

O comportamento do K-means é importante para o entendimento dos resultados obtidos pela política CH. Como o *dataset* $data$ é grande (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. Por outro lado, percebe-se que a opção de persistência CH mostra um tempo de recuperação mais eficiente em cenários de falha. Isto é, a diferença de tempo de execução entre cenários de falha com CH é a menor em comparação com as demais opções de persistência estáticas. A Figura 2 destaca essa observação através das sobrecargas de CF obtidas, que correspondem à diferença do resultado em um CF com relação a mesma execução sem falha.

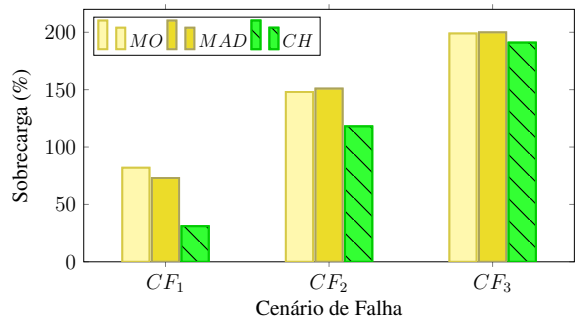

Figura 2. Sobrecargas de CF geradas por cenários de 5 iterações do K-means com políticas de persistência estáticas.

Em cenários de falhas, o desempenho das políticas estáticas é definido pelo comportamento das etapas de recuperação. As sobrecargas de CF exibidas na Figura 2 demonstram que as políticas de persistência em memória adicionam um processamento significativo conforme mais falhas são induzidas. A política CH, por outro lado, tem sobrecargas de CF consideravelmente menores.

Assim, a diferença de desempenho entre os cenários de falha 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 apresentou os menores níveis de sobrecarga. 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 à MO, entretanto, dificulta a consolidação das políticas de persistência estáticas nos testes realizados. Portanto, percebe-se que a alternativa de *checkpoint* do *dataset* inicial do K-means pode ser favorável apenas para casos 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.

4.3.2. Políticas de Monitoramento Dinâmicas

Nos testes com políticas dinâmicas, o K-means foi executado com base na política estática MO, em que o RDD $data$ é marcado para persistência em memória. Contudo, as políticas dinâmicas possuem a vantagem de observar *datasets* dentro da própria execução de uma aplicação. Desta forma, todos os *datasets* da Tabela 1 podem ser considerados para *checkpoint* através da DCA, de acordo com a política usada, mesmo quando uma parte do código é encapsulada. Os resultados obtidos com a DCA são exibidos pela Tabela 4.

Tabela 4. Resultados do K-means com políticas de monitoramento dinâmicas.

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

O tempo de execução obtido pelas políticas de *threshold* de *lineage* LT_3 e LT_{10} esclarecem a sensibilidade do K-means quanto à política de persistência CH. Com um *threshold* pequeno, muitos *checkpoints* são realizados – inclusive $data$. Assim, o desempenho é semelhante ao apresentado pela política estática CH, com tempos de execução impeditivos. Quando o limiar é aumentado para 10 dependências (LT_{10}), o desempenho da política possui uma grande alteração. O tempo de execução diminui significativamente, uma vez que o *dataset* $data$ deixa de ser considerado. Ainda assim, a política de monitoramento LT_{10} realiza salvamentos de *checkpoint* em cenários sem falha, que justificam o tempo excedente 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 de monitoramento dinâmicas. A falta de

informações sobre o fator MTBF bloqueia o início da etapa 2 da política, portanto nenhum *checkpoint* foi realizado. Por isso, a política obteve resultados satisfatórios, já que o cenário sem falha não é beneficiado por salvamentos de *checkpoint*.

Já em cenários com 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 fim de evidenciar essa conclusão, a Figura 3 mostra a sobrecarga de CF das políticas de monitoramento (LT e FA) e, para fins de comparação, também exibe a sobrecarga de CF da política de persistência MO.

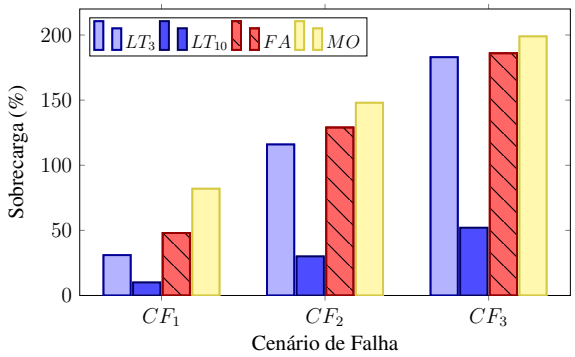

Figura 3. Sobrecargas de CF das políticas de monitoramento e da política de persistência MO.

Pode-se observar que as políticas FA, LT_3 e LT_{10} mantiveram uma sobrecarga de CF menor que a da política de persistência MO, conforme as falhas foram induzidas. A política LT_{10} destacou-se com a menor sobrecarga de CF dentre todas as demais. Esse comportamento se deve ao seu cenário sem falha – *baseline* para a sobrecarga de CF – apresentar um tempo de execução maior que FA e MO. Por outro lado, a baixa sobrecarga apresentada por LT_{10} em cenários de falha revela que reprocessamentos são mais custosos que a leitura de *checkpoints* em partes da aplicação. De acordo com os resultados da política de persistência CH, o salvamento do *dataset* $data$ é altamente custoso e inviabiliza o uso de *checkpoint*. Porém, o salvamento de outros *datasets* mostrou um baixo custo e uma sobrecarga de CF menor. Como exemplo, ao não escolher $data$ para *checkpoint*, a LT_{10} evitou o alto tempo de execução de LT_3 e CH.

A baixa sobrecarga de CF da política de monitoramento LT_{10} é pouco afetada pela quantidade de falhas, conforme aponta a Figura 3. A cada iteração, a *lineage* dos RDDs envolvidos tende a ser maior, exigindo um caminho maior a ser percorrido pelo procedimento de recuperação. 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.

Ao mesmo tempo, os resultados da Tabela 4 mostram um melhor tempo de execução da política FA. Assim, entende-se que a maioria dos *datasets* da execução possuem uma alta complexidade de processamento, que justifica a quebra da *lineage* e consequente recuperação via leitura do HDFS. Porém, alguns *datasets* também demandam tempo de escrita no HDFS, o que torna a política FA mais adequada em termos gerais. A Tabela 5 auxilia na compreensão dessa conclusão através da quantidade e do

custo médio de salvamento dos *checkpoints* estabelecidos pelas políticas FA e LT_{10} em cenários de falha. Além disso, a Tabela 5 exibe o percentual que o tempo de salvamento representa no tempo de execução total do respectivo cenário testado.

Tabela 5. *Checkpoints* com políticas dinâmicas em execuções com 5 iterações.

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_3	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_3	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_3	1,7	0,4%	4,5

Ainda que a quantidade de pontos usados pelo K-means seja grande o suficiente para gerar intrusividade quando o *dataset* $data$ (definição de todos os pontos) é salvo em *checkpoint* (vide resultados do CH), os *datasets* intermediários para o cálculo de centroides entre as iterações possuem dados resumidos. Consequentemente, o tamanho desses *datasets* é menor. Assim, o salvamento de *checkpoints* pelas políticas de monitoramento representam pouco do tempo de execução, em especial quando a política FA é utilizada.

5. Considerações Finais

Configurações estáticas limitam o desempenho e a confiabilidade da técnica de *checkpoint*, sobretudo em sistemas que trabalham com grandes quantidades de dados. Neste trabalho, utilizamos e validamos a aplicação de uma solução dinâmica para o estabelecimento de *checkpoints* no Apache Spark: a *Dynamic Configuration Architecure* (DCA). Nos testes realizados através do *benchmark* K-means, verificou-se que o desempenho obtido pela técnica de *checkpoint* apresentou uma dependência do *dataset* escolhido. Esse comportamento oferece insegurança para a aplicação, sendo que o tempo de execução pode ser impeditivo de acordo com escolhas inapropriadas.

As políticas de monitoramento dinâmicas da DCA auxiliaram na busca por configurações eficientes de *checkpoint*. A política LT_{10} mostrou que o estabelecimento de *checkpoints* serviu para uma recuperação mais eficiente que o reprocessamento. Mas a dependência de um atributo estático (*threshold*) não forneceu de forma total o comportamento dinâmico pretendido pela DCA. Com a política LT_3, o problema do *threshold* estático foi melhor observado pois seu desempenho manteve-se similar ao da CH.

A política FA destacou-se pelo desempenho obtido nos cenários com e sem falha. 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. O bom desempenho da política FA em relação às políticas LT_3 e LT_{10} nos cenários de falha justificou-se pelo mesmo motivo dos cenários sem falha: a quantidade desnecessária e ineficiente (sem noção de custos de leitura e reprocessamento) de salvamentos com a política LT comprometeu seu desempenho.

Por fim, destaca-se que as soluções de *checkpoint* dinamicamente configurado em execuções no Spark conseguiram explorar a persistência de *datasets* inacessíveis via código pelo *HiBench*. Essa característica foi essencial para os testes explorados, a partir da variação de desempenho e de confiabilidade das diferentes políticas usadas. Tal variação apresentou resultados menos eficientes nos casos CH e LT_3 em cenários de falha. Todavia, a presença de *checkpoints* auxiliou em recuperações mais rápidas, o que torna o uso desta técnica essencial em sistemas e plataformas de baixa confiabilidade.

Em projetos futuros, novos testes serão realizados a partir de *benchmarks* com diferentes propósitos considerando alterações no número de partições em que os RDDs do Spark são divididos. Investigações mais profundas sobre a literatura e comparações com outras soluções dinâmicas também devem ser conduzidas. Além disso, aplicações reais serão submetidas a testes com a DCA, para que a confiabilidade e o desempenho proporcionados pela arquitetura possam ser validados em um ambiente de produção.

Referências

Cardoso, P. V. and Barcelos, P. P. (2018a). Dynamic checkpoint architecture for reliability improvement on distributed frameworks. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 261–263. IEEE.

Cardoso, P. V. and Barcelos, P. P. (2018b). Validation of a dynamic checkpoint mechanism for apache hadoop with failure scenarios. In *2018 IEEE 19th Latin-American Test Symposium (LATS)*, pages 1–6. IEEE.

Egwutuoha, I. P., Levy, D., Selic, B., and Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326.

Foundation, A. S. (2019). "Apache Spark: Quick Start". `https://spark.apache.org/docs/2.4.1/rdd-programming-guide.html`. Novembro.

Karau, H. and Warren, R. (2017). *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. "O'Reilly Media, Inc.".

Laprie, J.-C. (1985). Dependable computing and fault tolerance: Concepts and terminology. In *25th International Symposium on Fault-Tolerant Computing*, page 2. IEEE.

Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). MLlib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241.

Verma, J. P. and Patel, A. (2016). Comparison of mapreduce and spark programming frameworks for big data analytics on HDFS. *International Journal of Computer Science and Communication*, 7(2):80–84.

White, T. (2015). *Hadoop: The Definitive Guide, 4th Edition*. "O'Reilly Media, Inc.".

Yan, Y., Gao, Y., Chen, Y., Guo, Z., Chen, B., and Moscibroda, T. (2016). Tr-spark: Transient computing for big data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 484–496. ACM.

Zhu, W., Chen, H., and Hu, F. (2016). ASC: Improving spark driver performance with automatic spark checkpoint. In *2016 18th International Conference on Advanced Communication Technology (ICACT)*, pages 607–611. IEEE.