

João Batista de Souza Neto

**Uma abordagem para Teste de Mutação de
programas de processamento de *Big Data***

Natal-RN

2019

João Batista de Souza Neto

**Uma abordagem para Teste de Mutação de programas
de processamento de *Big Data***

PPgSC – Programa de Pós-Graduação em Sistemas e Computação

DIMAp – Departamento de Informática e Matemática Aplicada

CCET – Centro de Ciências Exatas e da Terra

UFRN – Universidade Federal do Rio Grande do Norte

Orientador: Martin A. Musicante

Coorientadora: Anamaria Martins Moreira

Coorientadora: Genoveva Vargas-Solar

Natal-RN

2019

Resumo

O crescimento do volume de dados gerado pela sociedade, sua produção contínua e em larga escala e sua heterogeneidade levaram ao desenvolvimento do conceito de *Big Data*. O acesso cada vez mais aberto a esses dados apresenta desafios na coleta, armazenamento e, sobretudo, no seu processamento e análise, que exigem importantes recursos de computação e condições de execução adaptada. Diferentes modelos de programação paralela e sistemas foram propostos para o processamento de Big Data, como o modelo MapReduce e sua implementação no sistema Hadoop, e os sistemas Dryad, Nephle e o Apache Spark, além de outros. Esses sistemas, projetados para arquiteturas do tipo cluster, oferecem ambientes paralelos de programação e execução que ocultam dificuldades técnicas associadas (como tolerância a falhas e distribuição de dados, por exemplo) e permitem o foco nos aspectos algorítmicos do processamento de dados. Independentemente do modelo e sistema adotado, aplicações de processamento de Big Data precisam ser testadas e avaliadas, principalmente levando em consideração os custos que envolvem a execução nesse contexto. Entretanto, a área de testes em Big Data ainda é nova, possuindo poucos trabalhos que buscam aplicar técnicas sistemáticas de teste. Esta proposta de tese de doutorado visa reduzir a lacuna existente na área de testes de aplicações de processamento de Big Data ao propor uma abordagem de *Teste de Mutação*. Esta é uma técnica de teste que busca simular defeitos em um programa ao inserir modificações em seu código, através da aplicação dos chamados operadores de mutação que determinam como a modificação é feita, de forma a criar diferentes versões deste, chamados de mutantes. Esses mutantes podem, então, ser utilizados tanto para avaliar um conjunto de testes, de modo a verificar quantos defeitos esse conjunto consegue identificar, quanto para projetar testes, de modo a criar testes que consigam identificar os defeitos simulados. Este trabalho propõe uma abordagem de teste de mutação baseado em um modelo que abrange as principais características de sistemas de processamento de Big Data e em defeitos que podem aparecer nesse contexto. Para identificar os tipos de defeitos que podem aparecer em programas de processamento de Big Data, foi realizada uma investigação sobre defeitos e problemas relacionados com o sistema Apache Spark. Esse estudo resultou no desenvolvimento de duas taxonomias. A primeira taxonomia agrupa e caracteriza problemas não-funcionais que afetam o desempenho de execução de uma aplicação Spark. A segunda taxonomia é focada em defeitos funcionais que afetam o comportamento de aplicações Spark. Uma definição de operadores de mutação e uma estratégia para a geração de mutantes é derivada desta segunda taxonomia, permitindo projetar uma abordagem para o teste de mutação de aplicações de processamento de Big Data.

Palavras-chaves: Big Data; Teste de Mutação; Apache Spark; Taxonomia; Operadores de Mutação.

Lista de ilustrações

Figura 2.1 – Atividades de Teste.	19
Figura 2.2 – Processo do Teste de Mutação (caixas em negrito são etapas automa- tizadas e caixas tracejadas são etapas manuais).	26
Figura 3.1 – Visão geral do processo de execução do modelos <i>MapReduce</i> no Apache Hadoop.	30
Figura 3.2 – Exemplo de aplicação de contagem de palavras no Hadoop MapReduce. . .	31
Figura 3.3 – Modelo de dados do <i>DryadLINQ</i>	35
Figura 3.4 – Exemplo de aplicação de contagem de palavras no DryadLINQ.	36
Figura 3.5 – Visão geral da execução de um programa <i>DryadLINQ</i>	37
Figura 3.6 – Processo de compilação e execução no Nephele/PACTs.	38
Figura 3.7 – Componentes de um PACT.	39
Figura 3.8 – Representação dos cinco Contratos de Paralelização de Entrada do Nephele/PACTs.	41
Figura 3.9 – Exemplo de aplicação de contagem de palavras no Nephele/PACTs. . .	43
Figura 3.10–Exemplo de aplicação de contagem de palavras no Apache Beam. . . .	48
Figura 3.11–Representação de transformação estreita e ampla.	52
Figura 3.12–Exemplo de aplicação Spark de contagem de palavras.	53
Figura 3.13–Visão geral do cluster.	54
Figura 4.1 – Taxonomia de Problemas de Desempenho.	60
Figura 4.2 – Ilustração do problema de não utilizar variáveis de transmissão. . . .	63
Figura 4.3 – RDD Não Pré-Particionado.	65
Figura 4.4 – Particionamentos Diferentes.	65
Figura 4.5 – Exemplo de redução com <i>groupByKey</i>	66
Figura 4.6 – Exemplo de redução com <i>reduceByKey</i>	66
Figura 4.7 – Agregação com mudança de tipo utilizando <i>reduceByKey</i>	67
Figura 4.8 – Agregação com mudança de tipo utilizando <i>aggregateByKey</i>	68
Figura 4.9 – repartition versus coalesce.	68
Figura 4.10– <i>Speedups</i> dos experimentos de Gerenciamento de Recursos do Cluster. .	76
Figura 4.11– <i>Speedups</i> dos experimentos de Particionamento de Dados.	77
Figura 4.12– <i>Speedups</i> dos experimentos de Transmissão de Dados.	78
Figura 4.13– <i>Speedups</i> dos experimentos de Persistência de Dados.	79
Figura 4.14– <i>Speedups</i> dos experimentos de RDD Não Pré-Particionado.	80
Figura 4.15– <i>Speedups</i> dos experimentos de Particionamentos Diferentes.	81
Figura 4.16– <i>Speedups</i> dos experimentos de <i>groupByKey</i> versus <i>reduceByKey</i>	82
Figura 4.17– <i>Speedups</i> dos experimentos de <i>reduceByKey</i> versus <i>aggregateByKey</i> . . .	82
Figura 4.18– <i>Speedups</i> dos experimentos de <i>repartition</i> versus <i>coalesce</i>	83

Figura 5.1 – Taxonomia de Defeitos Funcionais.	86
Figura 5.2 – Exemplo de aplicação Spark que faz uma análise em arquivos de <i>Log</i> . . .	90
Figura 5.3 – Exemplo de aplicação Spark que estima o valor de π	91
Figura 5.4 – Exemplo de aplicação Spark que computa <i>n-grams</i> e calcula suas frequências.	91
Figura 5.5 – Exemplo de aplicação Spark com transformações de conjuntos.	92
Figura 5.6 – Exemplo de aplicação Spark com operações de junção e ordenação. . .	92
Figura 5.7 – Exemplo de aplicação Spark com ações numéricas.	93
Figura 5.8 – Exemplo de aplicação Spark que utiliza um acumulador.	93
Figura 5.9 – Exemplo de implementação de um acumulador personalizado.	94
Figura 8.1 – Cronograma para as próximas atividades da pesquisa.	132

Lista de tabelas

Tabela 2.1 – Exemplos de operadores de mutação.	23
Tabela 3.1 – Comparação das interface de programação de sistemas de processamento de Big Data.	58
Tabela 4.1 – Perfis de configuração do cluster.	71
Tabela 4.2 – Aplicações executadas nos experimentos.	73
Tabela 4.3 – Resultados de desempenho agregados para cada perfil de configuração.	73
Tabela 4.4 – Resultados de tempo de desempenho para cada aplicação.	74
Tabela 4.5 – Resultados de entradas e saídas para cada aplicação.	75
Tabela 5.1 – Exemplos de defeitos e suas classificações segundo a Taxonomia de Defeitos Funcionais de Spark.	95
Tabela 6.1 – Descrição das representações do fluxo de dados de um programa de processamento de <i>Big Data</i>	101
Tabela 7.1 – Mapeamentos padrões para tipos de dados comuns.	123

Sumário

1	INTRODUÇÃO	8
1.1	Motivação	9
1.2	Objetivos	11
1.3	Metodologia	12
1.4	Organização do trabalho	14
2	TESTE DE SOFTWARE	16
2.1	Fundamentos	16
2.2	Atividades de Teste	19
2.3	Crterios de Cobertura	20
2.4	Teste de Mutação	21
2.5	Considerações Finais	26
3	SISTEMAS DE PROCESSAMENTO DE BIG DATA	28
3.1	Apache Hadoop	28
3.2	Dryad e DryadLINQ	32
3.2.1	Dryad	32
3.2.2	DryadLINQ	33
3.2.3	Execução	36
3.3	Nephele/PACTs	38
3.3.1	Nephele	38
3.3.2	PACTs	39
3.3.3	Compilação e Execução	44
3.4	FlumeJava e Apache Beam	44
3.4.1	Conceitos	45
3.4.2	Transformações	46
3.4.3	Avaliação Diferida, Optimização e Execução	47
3.5	Apache Spark	48
3.5.1	Operações	49
3.5.2	Variáveis Compartilhadas	53
3.5.3	Cluster Spark	54
3.5.4	Execução	55
3.6	Discussões	55
4	UMA TAXONOMIA DE PROBLEMAS DE DESEMPENHO PARA APACHE SPARK	59

4.1	Taxonomia	59
4.2	Experimentos	69
4.2.1	Ambiente de Execução	69
4.2.2	Metodologia	69
4.2.3	Conjuntos de Dados	70
4.2.4	Aplicações	72
4.2.5	Resultados	72
4.3	Discussões	74
5	UMA TAXONOMIA DE DEFEITOS FUNCIONAIS PARA APACHE SPARK	85
5.1	Taxonomia	85
5.2	Exemplos	90
5.3	Discussões	94
6	UM MODELO GERAL PARA PROGRAMAS DE PROCESSAMENTO DE BIG DATA	98
6.1	Fluxo de Dados	98
6.2	Conjunto de Dados e Transformações	102
6.2.1	Álgebra de Monoides	102
6.2.2	Conjuntos de Dados	104
6.2.3	Transformações	104
6.3	Exemplo	116
7	PROJETANDO OPERADORES DE MUTAÇÃO PARA PROGRAMAS DE PROCESSAMENTO DE BIG DATA	119
7.1	Classificação dos Operadores de Mutação	119
7.2	Operadores de Mutação de Fluxo de Dados	120
7.3	Operadores de Mutação para Transformações	122
7.4	Discussões	126
8	CONSIDERAÇÕES FINAIS E PRÓXIMAS ATIVIDADES	129
8.1	Contribuições	130
8.2	Próximas Atividades	131
	Referências	133

1 Introdução

O conceito tradicional da *World Wide Web* (ou simplesmente Web) vem sendo expandido para incluir relacionamentos entre quaisquer dispositivos capazes de se conectar à rede mundial. Esta nova dimensão da Web, chamada de Internet das Coisas (ou *Internet of Things, IoT*) (GUBBI et al., 2013) permite interações não apenas entre pessoas, mas também entre dispositivos (com diferentes graus de autonomia) e entre pessoas e dispositivos. A popularização das redes sociais, crescimento da IoT e diferentes fontes da indústria e pesquisas na academia têm provocado um enorme crescimento no volume de dados que circula pela Web. A captação e análise destes dados representa uma oportunidade para empresas e organizações interessadas em estudar o comportamento das entidades que produzem e consomem esses dados, incluindo o seu uso nos mais diversos fins. O crescimento do volume de dados, sua produção contínua e em larga escala e sua heterogeneidade levaram ao desenvolvimento do conceito de *Big Data*.

O desafio de pesquisas na área de Big Data é a elaboração de técnicas e ferramentas que sejam capazes de processar esses dados, de forma a acompanhar a variação do *volume* e *velocidade* de produção, considerando a *variedade* dos dados produzidos e a *veracidade* dos mesmos. Essas quatro características, aliadas ao possível *valor* obtido do processamento desses dados constituem as chamadas *Cinco Vês* do *Big Data* (MARR, 2015). O processamento e análise de grandes volumes de dados é uma área da Ciência da Computação que propõe novas técnicas e ferramentas que visam atender aos desafios propostos pelas *Cinco Vês* do *Big Data*. Para isto, novos modelos de representação e processamento de dados, assim como novos sistemas têm sido desenvolvidos nos últimos anos.

Diferentes modelos de programação paralela e distribuída, assim como sistemas, vêm sendo propostos para o processamento de Big Data nos últimos anos. O *MapReduce* (DEAN; GHEMAWAT, 2004) foi proposto pela *Google* como um modelo de programação para processamento paralelo e distribuído em uma arquitetura de *cluster* (grupo) de computadores que podia ser aplicado no processamento de grandes volumes de dados. Este modelo ganhou destaque por ter sido um dos primeiros a prover um modelo simplificado e escalável para o processamento de Big Data, de modo que detalhes complexos inerentes ao ambiente distribuído, como paralelização, distribuição dos dados, tolerância a falhas e balanceamento de carga, podiam ser abstraídos pelo desenvolvedor (DEAN; GHEMAWAT, 2004). O modelo MapReduce ganhou popularidade com sua implementação de código aberto no sistema *Apache Hadoop* (HADOOP, 2019), que se tornou uma infraestrutura para uma série de sistemas para Big Data.

Apesar das vantagens do MapReduce, este também possuía uma série de limitações em relação ao seu desempenho de execução e ao seu modelo de programação rígido que não é adequado para uma série de análises e processamentos comuns no contexto de Big Data (KALAVRI; VLASSOV, 2013). Dessa forma, uma série de sistemas e modelos foram desenvolvidos com o intuito de resolver as limitações existentes no MapReduce. Entre esses sistemas estão o *Dryad* (ISARD et al., 2007) e *DryadLINQ* (YU et al., 2008), ambos desenvolvidos pela *Microsoft*, o sistema *Nephele* (WARNEKE; KAO, 2009) e o modelo *PACTs* (BATTRE et al., 2010), o *FlumeJava* (CHAMBERS et al., 2010), também desenvolvido pela Google, e a sua implementação de código aberto no *Apache Beam* (BEAM, 2016), e, por último, o *Apache Spark* (ZAHARIA et al., 2010). Todos esses sistemas oferecem ambientes de programação paralela e distribuída em *cluster* de computadores que ocultam dificuldades técnicas associadas com o ambiente, como tolerância a falhas e distribuição de dados, por exemplo, permitindo que desenvolvedores se foquem nos aspectos algorítmicos do processamento de dados.

1.1 Motivação

Os desafios que envolvem o processamento e análise de Big Data e a quantidade significativa de recursos que são necessários para a sua viabilidade, como a alocação de computadores para armazenamento e processamento em um cluster, fazem com que problemas em uma aplicação possam ter efeitos potencialmente adversos (GARG; SINGLA; JANGRA, 2016). Por esse motivo, é essencial que uma aplicação de processamento de Big Data seja *verificada* e *validada* antes de ser colocada em produção, uma vez que falhas e defeitos nessa aplicação podem implicar em grandes prejuízos. Uma importante técnica de verificação e validação e a mais aplicada na indústria é o *teste de software*.

O processo de testar um software consiste em verificá-lo dinamicamente com a intenção de ver se este se comporta da maneira esperada (ABRAN et al., 2004). A principal finalidade desse processo é procurar por defeitos no software (MYERS et al., 2004), fazendo com que o teste esteja diretamente ligado a qualidade do mesmo. Testar aplicações de *Big Data* esbarra não só nos desafios inerentes da área (Cinco Vês), como também na falta de experiência de desenvolvedores e engenheiros de teste em testar apropriadamente aplicações do tipo (MITTAL, 2013; NACHIYAPPAN; JUSTUS, 2013). Isso cria uma série de oportunidades e desafios para a pesquisa na área de teste de Big Data, que deve levar em consideração todas as suas etapas: *pré-processamento*, que envolve o carregamento dos dados a partir de diferentes fontes; *processamento*, que envolve o tratamento, transformação e análise dos dados; e, por último, *extração e entrega dos dados*, que envolve validar o processamento feito nos dados e distribuir os resultados para todas as partes interessadas (GUDIPATI et al., 2013).

Nesse sentido, o teste de aplicações de processamento de Big Data é uma área de pesquisa em aberto que vem ganhando interesse nos últimos anos. Entretanto, esta é, ainda, uma área de pesquisa nova e que possui um número relativamente pequeno de trabalhos, como apontam (CAMARGO; VERGILIO, 2013b) e (MORÁN; RIVA; TUYA, 2019). Desses, poucos são os trabalhos que aplicam técnicas e critérios sistemáticos de teste (CAMARGO; VERGILIO, 2013a) e muitos ainda não foram apropriadamente validados (CSALLNER; FEGARAS; LI, 2011; MORÁN; RIVA; TUYA, 2015). Além disso, os autores em (MORÁN; RIVA; TUYA, 2019) mostram que pesquisas na área de teste de programas de processamento de Big Data tem se concentrado majoritariamente em programas no estilo MapReduce. Para outros sistemas e modelos, como o Apache Spark, Dryad e DryadLINQ, Nephele/PACTs e FlumeJava e Apache Beam, é possível encontrar bibliotecas e *frameworks* que possibilitam o teste de unidade de aplicações. Entretanto, considerando o limite do nosso conhecimento, ainda não existem trabalhos que abordam o teste sistemático de programas de processamento de Big Data que não sejam focados apenas em programas MapReduce, o que indica uma limitação e uma oportunidade de pesquisa na área.

Uma importante técnica de teste de software é o *teste de mutação* (DEMILLO; LIPTON; SAYWARD, 1978), que é uma técnica de testes baseados em defeitos. Seu processo consiste em criar variantes de um programa original, chamados de *mutantes*, a partir da simulação de defeitos comuns feitas por mudanças sintáticas no programa, e projetar testes que consigam distinguir os mutantes do programa original (AMMANN; OFFUTT, 2017). Nesta abordagem, um teste deve identificar que o resultado obtido com um mutante é diferente do resultado obtido com o programa original. Os mutantes são criados a partir da inserção de defeitos no programa original. Esses defeitos são pequenas modificações sintáticas e variações no código feitas a partir de regras com padrões de mudança. Essas regras, chamadas de *operadores de mutação*, são projetadas para simular defeitos e enganos de programação comuns, formando a base do teste de mutação.

Teste de mutação pode ser utilizado em dois contextos diferentes. O primeiro, consiste em utilizá-lo como um critério para projetar testes, de modo que testes devem ser projetados para distinguir os mutantes do programa original (AMMANN; OFFUTT, 2017). O segundo contexto em que teste de mutação é aplicado, é para medir a qualidade de um conjunto de testes já projetado ou para avaliar outros critérios e técnicas de teste (OFFUTT; MA; KWON, 2004). Uma vez que teste de mutação possui a característica de simular defeitos em um software, é esperado que testes que atendam o critério também consigam identificar defeitos reais quando estes ocorrerem e forem similares aos defeitos simulados. Diferentes trabalhos vem comprovando a efetividade do teste de mutação ao compará-lo de forma empírica com outros critérios e técnicas de teste, como os trabalhos apresentados em (FRANKL; WEISS; HU, 1997), (OFFUTT et al., 1996) e (WALSH, 1985). Dessa forma, o teste de mutação se estabeleceu como uma referência na

área de teste de software.

1.2 Objetivos

Levando em consideração a lacuna existente na área de teste de programas de processamento de Big Data apontada na seção anterior e as vantagens do Teste de Mutação em relação a outros critérios de teste, além do seu uso como parâmetro de qualidade para avaliação de testes de maneira geral, este trabalho tem como alvo de pesquisa investigar se *é possível ou não aplicar teste de mutação em programas de processamento de Big Data*.

Para responder essa questão de pesquisa, levamos em consideração os requisitos necessários para se criar uma abordagem de teste de mutação. Segundo (DELAMARO; OFFUTT; AMMANN, 2014), o teste de mutação deve *(i)* ser baseado em uma caracterização bem fundamentada de defeitos e *(ii)* na estrutura de uma linguagem de programação ou tecnologia alvo. Dessa forma, operadores de mutação podem ser projetados para simular defeitos e enganos comuns de programação que são passíveis de acontecer em um cenário real (AMMANN; OFFUTT, 2017), algo que caracteriza o teste de mutação como uma técnica de teste baseado em defeitos. Além disso, ter um maior entendimento sobre os tipos de defeitos que podem aparecer em um software ajuda a mitigar os seus efeitos e a estabelecer estratégias para evitá-los ou minimizá-los. Com base nisso, nós definimos como objetivos desse trabalho:

- *Definir uma taxonomia de defeitos no contexto de programas de processamento de Big Data:* pretendemos investigar os tipos de defeitos e problemas que podem aparecer no contexto de programas de processamento de Big Data de modo a construir uma taxonomia que agrupe e caracterize esses defeitos. Para esse estudo, escolhemos como alvo o Apache Spark devido ao destaque que esse vem ganhado nos últimos.
- *Definir um modelo geral para programas de processamento de Big Data:* uma vez que nosso trabalho busca preencher uma lacuna existente na área de teste de programas de processamento de Big Data, pretendemos criar um modelo que generaliza as principais características e operações que são encontradas em programas do tipo. Com base nesse modelo, podemos criar uma abordagem que não atende apenas uma única tecnologia, mas que pode ser aplicada em outros que se encaixem no modelo;
- *Projetar operadores de mutação para programas de processamento de Big Data:* pretendemos utilizar a taxonomia de defeitos e o modelo de generalização de programas de processamento de Big Data desenvolvidos durante este trabalho como guias para projetar operadores de mutação que formam a base da nossa abordagem de teste de mutação.

1.3 Metodologia

Para atender os objetivos da nossa pesquisa, este trabalho inicia com uma investigação sobre os defeitos que podem surgir no contexto de programas de processamento de Big Data que utilizam o Apache Spark. O método para a realização desse estudo tomou como base outros estudos que visaram identificar e classificar defeitos de softwares em diferentes contextos. Em (NAKAMURA; HOCHSTEIN; BASILI, 2006), é apresentada uma abordagem para identificação e classificação de defeitos em domínios específicos utilizando um método de inspeção em códigos e análise de seus históricos de versões. Essa abordagem possui três atividades principais: *Suporte*, que consistem em atividades que dão apoio à análise, como estudo da literatura e desenvolvimento de ferramentas e heurísticas; *Análise*, que é a atividade mais importante do processo que consiste em selecionar um código, analisá-lo utilizando diretivas e heurísticas, que incluem a análise de seu histórico de versões, com o intuito de identificar, documentar e classificar defeitos, além de uma etapa final de desenvolvimento de hipóteses sobre como mitigar esses defeitos; e *Verificação*, que inclui atividades de validação dos defeitos identificados através de experimentos, entrevistas e *surveys*.

A abordagem de (NAKAMURA; HOCHSTEIN; BASILI, 2006) foi adaptada em (CAMARGO; VERGILIO, 2013a) para um estudo de identificação e classificação de defeitos em programas MapReduce. Nele, foram excluídas as atividades de análise do histórico do código devido a dificuldades em encontrar repositórios de projetos que utilizam MapReduce com histórico de versões. Nosso trabalho adaptou a metodologia apresentada em (NAKAMURA; HOCHSTEIN; BASILI, 2006) e (CAMARGO; VERGILIO, 2013a) para um processo iterativo e incremental visando construir uma taxonomia de defeitos para programas Apache Spark. Assim como em (CAMARGO; VERGILIO, 2013a), nós não analisamos diferentes versões de códigos porque não conseguimos encontrar repositórios públicos com projetos utilizando Apache Spark que possuíssem histórico de versões. Além disso, expandimos nossas fontes de análise, não limitando-se apenas ao código, mas também defeitos que eram retratados na literatura, quando existissem. As etapas do nosso estudo são descritas a seguir:

1. *Entendimento do domínio*: nesta fase, vamos nos certificar de que o domínio de estudo, ou seja, Apache Spark, foi entendido de forma abrangente para conduzir as próximas fases, o que inclui o estudo de artigos relacionados, livros didáticos e desenvolvimento de aplicações.

As fases seguintes são executadas de maneira iterativa e incremental.

2. *Seleção de uma fonte de estudo*: nesta fase, selecionamos uma fonte de estudo para analisar. Essa fonte pode ser projetos ou códigos que utilizam Spark, além da literatura sobre Spark, que pode abordar problemas relacionados a Spark.

3. *Análise da fonte de estudo*: nesta fase, nós examinamos a fonte de estudos para procurar por defeitos ou possíveis defeitos. Isso é feito através de uma análise manual do código ou projeto, ou leitura detalhada da literatura em questão, assim como a reprodução de códigos e experimentos desta fonte.
4. *Documentação de defeitos*: nesta fase, os resultados da análise, ou seja, os defeitos encontrados, serão documentados de forma a construir um conjunto de conhecimentos (*body of knowledge*) sobre defeitos em aplicações Spark. A documentação foi feita de forma similar a (NAKAMURA; HOCHSTEIN; BASILI, 2006), de modo que os documentos continham informações sobre a descrição e localização do defeito, descrição dos seus efeitos e comentários sobre como solucioná-lo.
5. *Classificação de defeitos*: nesta fase, os defeitos documentados são classificados em categorias. Com isso, pretendemos obter conhecimentos e identificar padrões em defeitos (NAKAMURA; HOCHSTEIN; BASILI, 2006). Esperamos construir um esquema de classificação abrangente agrupando falhas de acordo com informações comuns do contexto, descrições e consequências. As classificações e falhas apresentadas em (MORÁN; RIVA; TUYA, 2014), (CHILLAREGE et al., 1992), (BRASS; GOLDBERG, 2004), (MARIANI, 2003) e (CAMARGO; VERGILIO, 2013a) serviram como base para criar um esquema de classificação para defeitos em programas Spark.
6. *Refinar a classificação*: uma vez que estamos construindo a taxonomia de maneira iterativa, esperamos ter, ao final de uma iteração, mais conhecimentos sobre defeitos em programas Spark que na iteração anterior. Dessa forma, fazemos melhorias na classificação sempre que possível.

Seguindo essa abordagem, iniciamos nosso trabalho com um estudo abrangente sobre Apache Spark, o que incluiu uma análise de seus principais artigos (ZAHARIA et al., 2010) e (ZAHARIA et al., 2012), sua documentação oficial (SPARK, 2019), além de outras fontes na literatura, como os livros (GANELIN et al., 2016) e (KARAU; WARREN, 2017). Esse estudo envolveu o desenvolvimento de aplicações Spark para ter conhecimento prático na área. Durante esse processo, identificamos que existia uma preocupação recorrente entre os autores com relação ao desempenho de execução das aplicações. Uma vez que Spark é executado em um ambiente de cluster de computadores que leva em consideração custos em relação ao tempo e recursos utilizados, ter uma aplicação eficiente se torna uma preocupação maior. Por conta disso, parte das fontes estudadas, além de outros artigos, se dedicam a discutir problemas que influenciam o desempenho de execução de aplicações Spark, que podem ir de questões na configuração do cluster a problemas de projeto da aplicação. Por esta razão, decidimos expandir o nosso estudo ao analisar dois aspectos:

problemas que afetam o desempenho de execução da aplicação e defeitos funcionais na aplicação.

No primeiro caso, um problema no desempenho na aplicação pode não conter nenhum defeito funcional, ou seja, quando a aplicação é finalizada, seu resultado final está correto em relação àquilo que era esperado. Entretanto, esse problema pode afetar o desempenho consideravelmente, de modo que a aplicação consome mais recursos do cluster e leva mais tempo para ser finalizada. No segundo caso, um defeito funcional pode fazer com que o resultado final da aplicação não seja de acordo com o que é esperado, o que pode ser independente do desempenho de execução da mesma. Como resultado desse estudo, construímos duas taxonomias, uma contendo os problemas de desempenho e outra contendo defeitos funcionais para aplicações Spark.

A segunda parte do processo de criação dos operadores de mutação da nossa abordagem envolveu o desenvolvimento de um modelo de generalização para programas de processamento de Big Data. A criação desse modelo levou em consideração características comuns em relação ao fluxo de dados e operações em sistemas como o Apache Spark (ZAHARIA et al., 2010), Hadoop MapReduce (DEAN; GHEMAWAT, 2004), FlumeJava (CHAMBERS et al., 2010), Nephela/PACTs (BATTRÉ et al., 2010) e o Dryad e DryadLINQ (YU et al., 2008). Esse modelo foi definido a partir de dois formalismos, um formalismo baseado em *Redes de Petri* (MURATA, 1989) e em *Álgebra de Monoídes* (FEGARAS, 2017).

A partir da taxonomia de defeitos funcionais para Apache Spark e do modelo de generalização para programas de processamento de Big Data desenvolvidos, projetamos operadores de mutação. Os operadores de mutação foram projetados de modo a refletir os defeitos descritos na taxonomia e a explorar a estrutura de programas de processamento de Big Data, preenchendo assim os requisitos para a criação de uma abordagem de teste de mutação levantados por (DELAMARO; OFFUTT; AMMANN, 2014).

1.4 Organização do trabalho

O restante desta proposta de tese de doutorado está organizado como segue:

Capítulo 2 – Teste de Software: este capítulo apresenta os principais conceitos sobre Teste de Software e Teste de Mutação que fundamentam este trabalho;

Capítulo 3 – Sistemas de Processamento de Big Data: este capítulo faz uma introdução aos sistemas de processamento de Big Data estudados neste trabalho: Hadoop MapReduce, Dryad e DryadLINQ, PACTs/Nephela, FlumeJava e Apache Beam, e o Apache Spark. Ao final deste capítulo, é feita uma discussão acerca das similaridades entre esses sistemas;

Capítulo 4 – Uma Taxonomia de Problemas de Desempenho para Apache Spark: este capítulo apresenta a taxonomia de problemas de desempenho para Apache Spark desenvolvida a partir de um estudo sobre defeitos e problemas no contexto de Apache Spark. Também é apresentado os resultados de um experimento que corrobora a taxonomia;

Capítulo 5 – Uma Taxonomia de Problemas Funcionais para Apache Spark: este capítulo apresenta a taxonomia de defeitos funcionais para Apache Spark. Os defeitos apresentados na taxonomia são ilustrados através de exemplos;

Capítulo 6 – Um Modelo Geral para Programas de Processamento de Big Data: este capítulo apresenta um modelo de generalização para programas de processamento de Big Data desenvolvido a partir de características comuns em sistemas utilizados no processamento de Big Data;

Capítulo 7 – Projetando Operadores de Mutação para Programas de Processamento de Big Data: este capítulo apresenta os operadores de mutação para o teste de mutação de programas de processamento de Big Data que foram projetados a partir da taxonomia de defeitos funcionais para Apache Spark e do modelo de generalização para programas de processamento de Big Data;

Capítulo 8 – Considerações Finais e Próximas Atividades: este capítulo conclui esta proposta de tese de doutorado e lista as próximas atividades para a tese, assim como apresenta o seu cronograma.

2 Teste de Software

Este capítulo faz uma introdução aos principais conceitos sobre *Teste de Software* e *Teste de Mutação*. Iniciamos com uma apresentação dos fundamentos de teste de software na Seção 2.1 a partir da descrição da terminologia e conceitos básicos. Em seguida, fazemos uma apresentação das principais atividades do processo de teste de software na Seção 2.2. Depois, falamos sobre os conceitos de critérios de cobertura na Seção 2.3 e apresentamos o *teste de mutação* na Seção 2.4. Finalizamos com algumas considerações finais na Seção 2.5.

2.1 Fundamentos

Teste de software é o processo de executar um software com o intuito de verificar se este se comporta da maneira esperada (ABRAN et al., 2004). Em (MYERS et al., 2004), teste é definido como o processo de executar um software com a finalidade de encontrar erros. Com estas definições, é possível ver que o teste de software é uma atividade importante do processo de desenvolvimento e que está diretamente ligada à qualidade do software. O teste é o principal mecanismo de validação e verificação utilizado pela indústria e uma importante área de estudo dentro da Engenharia de Software.

A seguir, será apresentada uma série de termos que são importantes no contexto de teste de software, em geral, e neste trabalho. Existem diferentes terminologias relacionadas a teste de software na literatura, neste trabalho são adotados os termos apresentados em (AMMANN; OFFUTT, 2017). Dois termos importantes são validação e verificação:

Definição 2.1.1 (Validação) *É o processo de avaliar o software no final do seu desenvolvimento para garantir que ele atende às necessidades para as quais ele foi desenvolvido.*

Definição 2.1.2 (Verificação) *É o processo de determinar se os resultados de uma determinada fase do processo de desenvolvimento de software atendem ao que foi planejado na fase anterior.*

Validação e verificação são dois termos que costumam ser confundidos, mas a verificação é, geralmente, um processo mais técnico que requer conhecimentos sobre os artefatos do software, requisitos e especificações. Já a validação, requer conhecimentos sobre o domínio da aplicação para o qual o software foi desenvolvido. Outra distinção que se faz necessária é a entre os termos defeito, erro e falha:

Definição 2.1.3 (Defeito (*Fault*)) *Um defeito estático no código do software que pode resultar em algum problema quando executado.*

Definição 2.1.4 (Erro (*Error*)) *Um estado interno incorreto que é causado por um defeito no software.*

Definição 2.1.5 (Falha (*Failure*)) *É a externalização do erro. Um comportamento incorreto do software com relação ao que foi requisitado ou o que é esperado.*

As definições de defeito, erro e falha são importantes porque ajudam a distinguir os conceitos de *teste* e de *depuração* (*debugging*):

Definição 2.1.6 (Teste) *É o processo de avaliar um software observando a sua execução.*

Definição 2.1.7 (Falha de Teste) *É a execução que resulta em uma falha, ou seja, um comportamento inesperado.*

Definição 2.1.8 (Depuração (*Debugging*)) *É o processo de encontrar um defeito no software dada uma falha na execução.*

Mesmo com a existência de defeitos, nem toda a execução do software resultará em uma falha. Para que uma falha possa ser observada, são necessárias quatro condições: (i) o código (local) que possui o defeito deve ser executado (*Alcançabilidade*); (ii) é necessário que a execução do defeito ocasione um erro, infectando o estado interno do software (*Infecção*); (iii) é necessário que o estado infectado resulte em alguma falha do software (*Propagação*); e (iv) o testador deve ser capaz de observar a parte incorreta do estado final do programa (*Revelabilidade*). O conjunto das quatro condições necessárias para que um defeito ocasione uma falha é conhecido como modelo “RIPR” ou modelo defeito/falha (AMMANN; OFFUTT, 2017). Este modelo é importante para análises estáticas do software e também é utilizado no *Teste de Mutação*, que será apresentado mais à frente.

Definição 2.1.9 (Alcançabilidade (*Reachability*)) *O local ou locais que contém defeitos no programa devem ser alcançados.*

Definição 2.1.10 (Infecção (*Infection*)) *Após a execução do local que possui um defeito, o estado do programa deve ser incorreto.*

Definição 2.1.11 (Propagação (*Propagation*)) *O estado infectado deve causar alguma saída incorreta ou o estado final do programa deve ser incorreto.*

Definição 2.1.12 (Revelabilidade (*Revealability*)) *O testador deve ser capaz de observar a porção incorreta do estado final do programa.*

Um importante conceito é o de *caso de teste*, que, de forma geral, representa uma situação a ser testada no software. Um caso de teste inclui as condições necessárias para a execução do teste, os seus valores de entrada e os seus resultados esperados.

Definição 2.1.13 (Valores de Caso de Teste) *São os valores de entrada necessários para completar alguma execução do software sob teste.*

Definição 2.1.14 (Resultados esperados) *É o resultado que será produzido durante a execução do teste se, e somente se, o programa satisfaz seu comportamento pretendido.*

Definição 2.1.15 (Caso de Teste) *É um conjunto de artefatos que são necessários para uma completa execução e avaliação do software sob teste em uma determinada situação. Esses artefatos incluem os valores de caso de teste e resultados esperados, além das condições necessárias para que o software possa ser executado na situação a ser testada.*

Além dos casos de teste, um outro elemento fundamental no teste de software é o mecanismo que avalia os resultados do software para verificar se um teste foi bem sucedido ou não, ou seja, se houve a ocorrência de falhas. Esses mecanismos são conhecidos como *oráculos de teste* e têm o objetivo de verificar a correta execução do software dada uma certa entrada.

Definição 2.1.16 (Oráculo de teste) *É o mecanismo que determina se um programa se comportou corretamente em um dado teste, resultando no veredicto “passou” ou “falhou”.*

Os componentes no caso de teste e oráculo de teste formam uma realização do modelo RIPR uma vez que esses são projetados para procurar por defeitos no software, fornecendo as condições necessárias para que um defeito possa ser identificado. Um conjunto de casos de teste é denominado conjunto de testes ou suíte de testes.

Definição 2.1.17 (Conjunto de Testes) *É um conjunto de casos de teste.*

Projetar um conjunto de testes para um software não é uma tarefa trivial e, em várias situações, esbarra em problemas relacionados a como fornecer os valores de entrada para testar o software e como observar detalhadamente o seu comportamento.

Definição 2.1.18 (Observabilidade) *O quão fácil é observar o comportamento de um programa em termos de suas saídas, efeitos no ambiente e em outros componentes de hardware ou software.*

Definição 2.1.19 (Controlabilidade) *O quão fácil é fornecer ao programa as entradas necessárias, em termos de valores, operações e comportamentos.*

A observabilidade e a controlabilidade são importantes para determinar o quão testável é um software. Outras duas terminologias importantes no teste de software, que dizem respeito ao tipo do teste, são *teste funcional*, classicamente chamado de teste de caixa preta, e *teste estrutural*, classicamente chamado de teste de caixa branca.

Definição 2.1.20 (Teste Funcional) *O teste é derivado a partir de uma descrição externa do software, como a especificação ou requisitos, por exemplo.*

Definição 2.1.21 (Teste Estrutural) *O teste é derivado a partir do código fonte do software.*

2.2 Atividades de Teste

O teste de software envolve várias atividades que, geralmente, são realizadas pelo engenheiro de teste, um profissional encarregado por uma ou mais atividades de teste. De maneira geral, o processo de teste de software pode ser dividido em quatro grandes tipos de atividades: *Projeto de Teste*, *Automação de Teste*, *Execução de Teste* e *Avaliação de Teste*. Essas atividades podem ser vistas na Figura 2.1. Cada tipo de atividade requer diferentes níveis de habilidades e conhecimentos. Cada um dos tipos de atividade são descritos a seguir:

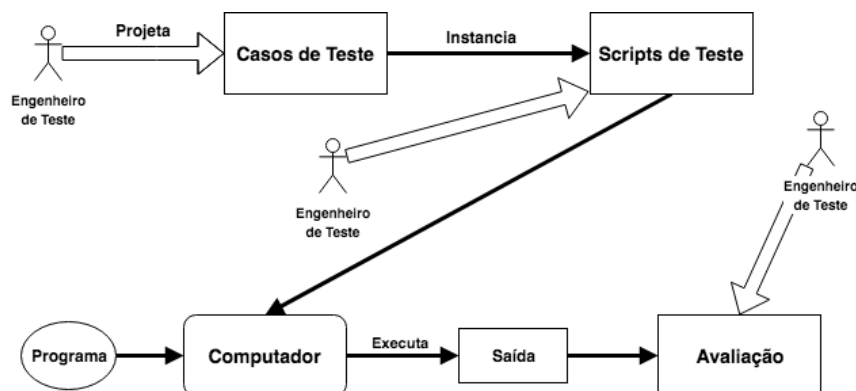


Figura 2.1 – Atividades de Teste.

Fonte: Adaptado de (AMMANN; OFFUTT, 2017)

1. *Projeto de Teste*: é o processo de projetar casos de teste que vão testar o software de maneira efetiva. Os testes podem ser projetados a partir de *critérios de cobertura* ou a partir de conhecimentos específicos sobre o domínio do sistema. Critérios de cobertura fornecem uma maneira sistemática de projetar testes com um nível satisfatório de qualidade;
2. *Automação de Teste*: é o processo de embutir os casos de teste (valores de entrada, resultados esperados, etc.) em *scripts* executáveis;
3. *Execução de Teste*: é o processo de executar os *scripts* de teste no software e registrar os seus resultados;
4. *Avaliação de Teste*: é o processo de avaliar os resultados dos testes e reportá-los para a equipe de desenvolvimento.

2.3 Critérios de Cobertura

Em um cenário ideal, o software deveria ser testado com todas as suas possíveis entradas, algo que é conhecido como *teste exaustivo*. Entretanto, uma vez que o número de possíveis entradas pode ser muito grande, possivelmente infinito, se torna inviável realizar um teste exaustivo. Por causa disso, é necessário projetar um conjunto de testes, não exaustivo, que consiga testar o software de uma maneira efetiva e viável. *Critérios de cobertura* são alguns dos mecanismos que ajudam a projetar conjuntos de testes mais efetivos e viáveis. Critérios de cobertura são definidos em termos de *requisitos de teste*:

Definição 2.3.1 (Requisito de teste) *Um requisito de teste é um elemento específico de um artefato de software que deve ser satisfeito ou coberto por algum caso de teste.*

Os requisitos de teste podem ser obtidos a partir de uma variedade de artefatos do software, como o código fonte, componentes do projeto, modelos da especificação e descrições do espaço de entrada, entre outros. Um critério de cobertura é apenas uma forma de se obter requisitos de teste de uma maneira sistemática:

Definição 2.3.2 (Critério de cobertura) *Um critério de cobertura é uma regra ou conjunto de regras que estabelecem requisitos de teste para um conjunto de testes.*

Para o engenheiro de testes, é importante saber a qualidade do seu conjunto de testes. É possível medir a qualidade de um conjunto de teste através de sua cobertura com relação a algum critério de cobertura.

Definição 2.3.3 (Cobertura) *Dado um conjunto de requisitos de teste TR para um critério de cobertura C , um conjunto de testes T satisfaz C se, e somente se, para cada requisito de teste $tr \in TR$, existe pelo menos um teste $t \in T$ que satisfaz tr .*

Definição 2.3.4 (Nível de cobertura) *Dado um conjunto de requisitos de teste TR e um conjunto de testes T , o nível de cobertura é a relação entre o número de requisitos satisfeitos por T com o número total de requisitos em TR .*

Uma vez que é possível obter requisitos de teste a partir de diferentes artefatos do software, existem vários tipos de critérios de cobertura que estabelecem requisitos de diferentes maneiras e a partir de diferentes artefatos. Em (AMMANN; OFFUTT, 2017), são apresentados os principais tipos de critério de cobertura: *cobertura de grafos*, *cobertura lógica*, *particionamento do espaço de entrada* e *testes baseados em sintaxe*. Este trabalho é baseado no critério de *Teste de Mutação*, que é um critério de testes baseado em defeitos (DELAMARO; MALDONADO; JINO, 2017).

2.4 Teste de Mutação

Proposto em (DEMILLO; LIPTON; SAYWARD, 1978), o *teste de mutação*, também conhecido como *análise de mutantes*, é uma técnica de testes baseados em defeitos que consiste em criar variantes de um programa original, chamados de *mutantes*, a partir da simulação de defeitos comuns que são inseridos através de pequenas modificações no programa original, e desenvolver testes que consigam distinguir os mutantes do programa original. O teste de mutação é fundamentado em duas hipóteses, a *hipótese do programador competente* e o *efeito de acoplamento* (DEMILLO; LIPTON; SAYWARD, 1978):

Definição 2.4.1 (Hipótese do Programador Competente) *Um programa P pronto para ser testado, quando desenvolvido por um programador competente, está correto ou próximo de estar correto.*

Definição 2.4.2 (Efeito de Acoplamento) *Testes que conseguem identificar pequenas variações (defeitos simples) no código do programa também são capazes de identificar defeitos mais complexos.*

A primeira hipótese implica que defeitos introduzidos por um programador experiente são causados, em sua maioria, por pequenos enganos ou desvios sintáticos por parte do programador. A segunda hipótese implica que defeitos mais complexos são resultados de uma série de defeitos mais simples juntos (acoplados). Consequentemente, ao

desenvolver testes que sejam capazes de identificar pequenos defeitos, também é possível identificar defeitos complexos.

Partindo dessas duas hipóteses, o teste de mutação estabelece os mutantes, programas com pequenas variações em relação ao original, como requisitos de teste, de modo que testes devem ser projetados para distinguir os mutantes de um programa original. Em outras palavras, um teste deve identificar que o resultado obtido com um mutante é diferente do resultado obtido com o programa original. Quando isso ocorre, é dito que o teste *matou* o mutante. Em certas situações, não é possível matar um mutante porque este, mesmo com alguma mudança, possui o mesmo comportamento do programa original, o que faz com que ambos não possam ser distinguidos, neste caso, o mutante é dito equivalente.

Definição 2.4.3 (Mutante) *Um mutante m é um programa que difere do programa original P devido a alguma mudança sintática ou pequena variação no código de P .*

Definição 2.4.4 (Mutante Morto) *Dado um mutante $m \in M$, em que M é um conjunto de mutantes de um programa original P , e um teste t , é dito que m foi morto por t se, e somente se, a saída que P provê para t é diferente da saída que m provê para t .*

Definição 2.4.5 (Mutante Equivalente) *Dado um mutante $m \in M$, é dito que m é equivalente a P se não existe um teste t capaz de matar m , ou seja, m possui o mesmo comportamento do programa original P .*

O processo de matar um mutante ou determinar que ele é equivalente nem sempre é trivial. Uma das formas de se analisar um mutante para determinar que ele é equivalente ou ver quais são as condições necessárias para matá-lo é através do modelo *RIPR*, uma vez que este modelo indica as condições necessárias para que um defeito possa ser identificado, ou seja, para que um mutante seja morto.

Mutantes são criados a partir da inserção de defeitos no programa, que são pequenas modificações no código fonte feitas a partir de regras com padrões de mudança. Essas regras, chamadas de *operadores de mutação*, quando aplicadas em um programa podem gerar um ou mais mutantes. Dada a definição de mutante feita anteriormente, também podemos definir um mutante como o resultado da aplicação de um operador de mutação em um programa. Alguns exemplos de operadores de mutação projetados para a linguagem *C* (RICHARD et al., 1989), além de linguagens com sintaxe parecida como *C++* e *Java*, podem ser vistos na Tabela 2.1.

Definição 2.4.6 (Operador de Mutação) *Uma regra que especifica um padrão de modificação ou desvio sintático para ser aplicado em um programa P .*

Sigla	Operador de Mutação	Descrição	Exemplo
OAAN	Substituição de operador aritmético	Substitui a ocorrência de um operador aritmético (+, -, *, /, %) por cada um dos outros operadores aritméticos	$a + b \rightarrow a * b$
OLLN	Substituição de operador lógico	Substitui a ocorrência de um operador lógico (&&,) por cada um dos outros operadores lógicos	$a \&\& b \rightarrow a b$
OLNG	Negação lógica	Faz a negação lógica de cada um dos operandos em uma expressão lógica, assim como a negação da expressão lógica como um todo	$a \&\& b \rightarrow a \&\& !b$
OCNG	Negação de contexto lógico	Faz a negação lógica de uma expressão que se encontra em um comando de repetição ou um comando condicional, com exceção do comando <i>switch</i>	$\text{if (exp) comando} \rightarrow \text{if (!exp) comando}$
CRCR	Substituição de constante	Faz a substituição de uma constante numérica por outras constantes numéricas	$a = b + 1 \rightarrow a = b + 0$
SSDL	Eliminação de comando	Elimina um comando	$c = a + b; \rightarrow ;$
ABS	Inserção de valor absoluto	Substitui o valor de uma variável em uma expressão aritmética por seu valor absoluto positivo e negativo	$a = 3 * b \rightarrow a = 3 * \text{abs}(b)$
UOI	Inserção de operador unário	Insere um operador unário (+, -) antes de uma variável em uma expressão aritmética	$a = 3 * b \rightarrow a = 3 * -b$

Tabela 2.1 – Exemplos de operadores de mutação.

Uma seleção criteriosa de operadores de mutação é fundamental para o sucesso na aplicação do teste de mutação uma vez que um bom operador de mutação pode gerar um mutante próximo de um defeito real. Dessa forma, se um teste consegue matar esse mutante, é esperado que este também seja capaz de identificar um defeito real similar. Por esse motivo, um operador de mutação deve ser fundamentado em uma boa caracterização de defeitos (DELAMARO; OFFUTT; AMMANN, 2014), de modo que ele deve ser projetado para “imitar” defeitos comuns ou para encorajar o testador a utilizar heurísticas comuns no desenvolvimento de testes (AMMANN; OFFUTT, 2017). Operadores de mutação são projetados, geralmente, para uma linguagem de programação específica levando em conta a sua estrutura ou sintaxe, o que faz com que teste de mutação possa ser classificado como um critério de cobertura baseado em sintaxe (AMMANN; OFFUTT, 2017). Pesquisas tem projetado operadores de mutação para diferentes linguagens de programação, como a linguagem C (RICHARD et al., 1989) e Java (MA et al., 2002). Além disso, operadores de mutação tem sido projetados para diferentes contextos, como para o teste de Programas Orientado a Aspectos (FERRARI; MALDONADO; RASHID, 2008), Especificações Formais (HASSINE, 2013), Serviços Web (LEE; OFFUTT, 2001) e SQL (SHAHRIAR; ZULKERNINE, 2008), além de vários outros, como mostrado em (JIA; HARMAN, 2011).

A partir da definição e seleção de um conjunto de operadores de mutação que, quando aplicados em um programa P , geram um conjunto de mutantes M , o critério de cobertura de teste de mutação estabelece como requisito de teste a morte de um mutante.

Definição 2.4.7 (Cobertura de Mutação) *Dado um conjunto de mutantes M de um programa P , para cada $m \in M$, matar m é um requisito de teste.*

Dependendo do número de operadores de mutação escolhidos e do tamanho do programa a ser testado, o número de mutantes gerados pode ser potencialmente grande, podendo facilmente chegar a milhares de mutantes. Esse fato faz com que o teste de mutação tenha um custo elevado. Por esse motivo, diferentes trabalhos se dedicam a pesquisar formas de reduzir os custos e complexidade do teste de mutação. Em (HOWDEN, 1982), o conceito tradicional de matar um mutante é suavizado ao considerar que um mutante m é morto por um teste t se o estado interno do programa é diferente imediatamente após a execução do local l em P , em que l representa o local onde foi feita a mutação. Essa definição difere do original porque exige apenas que um defeito seja *alcançável* e que este *infecte* o estado do programa, mas não exige que este seja *propagado* e *revelado*. Dessa forma, o resultado final de um mutante pode ser o mesmo do programa original, mas que mesmo assim este possa ser considerado morto. Esse novo conceito passou a ser conhecido como *matar fracamente* (*weakly kill*) um mutante e, em contrapartida, o conceito tradicional passou a ser conhecido como *matar fortemente* (*strongly kill*) um mutante. Matar fracamente um mutante pode reduzir o custo e complexidade do teste de mutação uma vez que este não exige que um programa e um mutante sejam executados de forma completa, mas apenas até o ponto em que foi feita a mutação (FERRARI; MALDONADO; RASHID, 2008). Os experimentos apresentados em (OFFUTT; LEE, 1991) mostram que matar fracamente os mutantes apresenta resultados próximos aos de matar fortemente em relação à quantidade de mutantes mortos, além de que o custo de matar fracamente é menor do que matar fortemente, o que faz matar fracamente um mutante ser uma alternativa viável. A partir disso, é possível definir novas formas para a cobertura de mutação:

Definição 2.4.8 (Mutante Fortemente Morto) *Dado um mutante $m \in M$ para um programa P e um teste t , é dito que m foi fortemente morto por t se, e somente se, a saída que P provê para t é diferente da saída que m provê para t .*

Definição 2.4.9 (Cobertura de Mutação Forte) *Dado um conjunto de mutantes M de um programa P , para cada $m \in M$, matar fortemente m é um requisito de teste.*

Definição 2.4.10 (Mutante Fracamente Morto) *Dado um mutante $m \in M$ para um programa P e um teste t , é dito que m foi fracamente morto por t se, e somente se, o estado interno da execução de P com t é diferente do estado interno da execução de m com t imediatamente após a execução de l .*

Definição 2.4.11 (Cobertura de Mutação Fraca) *Dado um conjunto de mutantes M de um programa P , para cada $m \in M$, matar fracamente m é um requisito de teste.*

A cobertura do teste de mutação como um requisito de testes pode ser determinada a partir da relação do número de mutantes mortos com o número total de mutantes, não considerando os mutantes equivalentes. Essa relação é conhecida como *escore de mutação* (*mutation score*) e pode ser utilizada como uma medida de qualidade para o conjunto de testes (DELAMARO; MALDONADO; JINO, 2017). O escore de mutação é calculado como segue (DELAMARO; MALDONADO; JINO, 2017):

$$ms(P, T) = \frac{DM(P, T)}{M(P) - EM(P)}$$

onde $ms(P, T)$ é o escore de mutação para um programa P e um conjunto de casos de teste T , $DM(P, T)$ é o número de mutantes do programa P mortos pelo conjunto de casos de teste T , $M(P)$ é o número de mutantes gerados a partir de P , e $EM(P)$ é o número de mutantes equivalentes a P .

Aplicar o teste de mutação envolve os seguintes passos: *geração dos mutantes*, *execução do programa original*, *execução dos mutantes* e a *análise dos mutantes* não mortos para determinar se estes são ou não equivalentes (DELAMARO; MALDONADO; JINO, 2017). Esse processo é mostrado na Figura 2.2, as etapas automatizadas estão representadas em caixas em negrito e as etapas manuais em caixas tracejadas. Dado um programa P , é criado um conjunto de mutantes M a partir da aplicação dos operadores de mutação selecionados em P . Em seguida, é gerado um conjunto de casos de teste T , que pode ser projetado com base nos mutantes em M ou a partir de outro critério a escolha do engenheiro de testes. Esse conjunto de testes T é então executado com o programa P . Caso os testes falhem, significa que é necessário fazer uma correção no programa P . Em caso contrário, o conjunto de casos de teste T é executado com os mutantes em M . A partir dos resultados dos testes, é feito o cálculo do escore de mutação (ms), com base no número de mutantes mortos e vivos. A partir desse passo, é necessário decidir se o processo do teste de mutação deve continuar ou não. Essa decisão pode ser feita com base no valor de ms , se este for 1, significa que todos os mutantes não equivalentes foram mortos pelo conjunto de testes T . Entretanto, nem sempre é viável matar todos os mutantes, dado o grande número de mutantes que podem ser gerados. Dessa forma, pode se definir, a critério do engenheiro de testes, um valor mínimo aceitável para o ms , geralmente próximo de 1, de modo que quando este for atingido, o conjunto de testes T pode ser considerado bom e o processo pode ser encerrado (DELAMARO; MALDONADO; JINO, 2017). Caso o valor de ms ainda não seja aceitável, é necessário fazer uma análise dos mutantes vivos. Essa análise pode resultar na identificação de mutantes equivalentes que não haviam sido identificados previamente, de modo que é possível fazer um novo cálculo do ms . Se a partir dessa nova análise o valor de ms ainda não for aceitável, é necessário fazer melhorias no conjunto de testes T e repetir o processo de execução do programa P .

e mutantes em M .

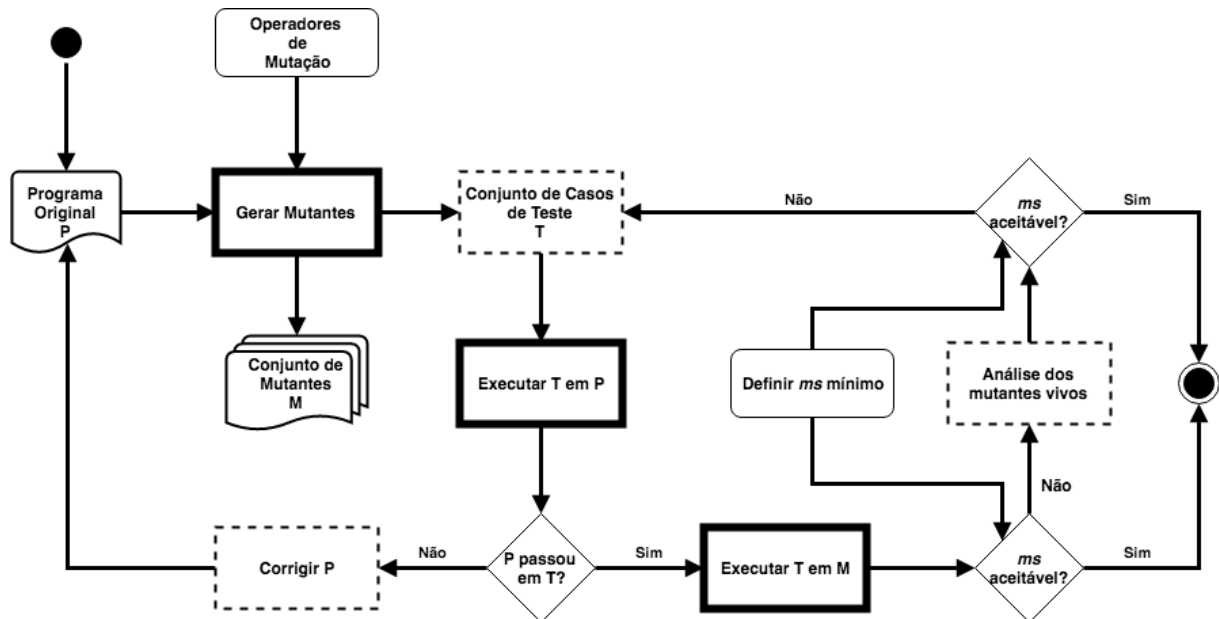


Figura 2.2 – Processo do Teste de Mutação (caixas em negrito são etapas automatizadas e caixas tracejadas são etapas manuais).

Fonte: Adaptado de (AMMANN; OFFUTT, 2017)

O teste de mutação é um dos critérios mais difíceis de satisfazer dado o grande esforço necessário para aplicar todas as suas etapas, como mostrado na Figura 2.2. Dado todo esse processo e, como dito anteriormente, a grande quantidade de mutantes que podem ser gerados, é fundamental que o teste de mutação seja suportado por uma ferramenta que automatize todo ou parte do processo, uma vez que é impraticável aplicá-lo manualmente. As etapas automatizadas são apresentadas com caixas em negrito na Figura 2.2. Diferentes ferramentas foram desenvolvidas através dos anos para dar suporte ao teste de mutação, como a *μJava* (MA; OFFUTT; KWON, 2005) e a *PIT*¹, ambas para a linguagem de programação *Java*, a *Nester*², para a linguagem *C#*, a *Milu* (JIA; HARMAN, 2008), para a linguagem *C* e o sistema *Proteum* (MALDONADO et al., 2001), que fornece uma família de ferramentas para teste de mutação em diferentes linguagens e contextos.

2.5 Considerações Finais

Este capítulo apresentou os conceitos básicos sobre teste de software e teste de mutação que são abordados neste trabalho. Ao longo dos anos, o teste de mutação tem sido investigado de forma ativa na área de pesquisa de teste de software (JIA; HARMAN, 2011). Estudos têm apontado o teste de mutação como um dos critérios mais eficientes

¹ <<http://www.pitest.org>>.

² <<http://www.nester.sourceforge.net>>.

em detectar defeitos em comparação a outros critérios, como os trabalhos apresentados em (FRANKL; WEISS; HU, 1997), (OFFUTT et al., 1996) e (WALSH, 1985). Esses resultados fazem com que o teste de mutação seja considerado uma referência na área de teste de software e que, frequentemente, seja utilizado como um padrão de qualidade para avaliar outros critérios, técnicas e conjuntos de teste de maneira geral (AMMANN; OFFUTT, 2017). Como dito anteriormente, o teste de mutação tem sido aplicado em diferentes linguagens de programação e diferentes contextos, como em (DELAMARO; MALDONADO; MATHUR, 2001), no contexto de integração de componentes, (FERRARI; MALDONADO; RASHID, 2008), no contexto de programas orientados a aspectos, (HASSINE, 2013), no contexto de máquinas abstratas de estado, e (SHAHRIAR; ZULKERNINE, 2008), no contexto de SQL, além de outros, como apontado em (JIA; HARMAN, 2011). Isso mostra que é possível aplicar o teste de mutação em diferentes contextos e tecnologias.

Segundo (DELAMARO; OFFUTT; AMMANN, 2014), trabalhos sobre teste de mutação se baseiam, de maneira geral, em *(i)* uma caracterização de defeitos e enganos de programação comuns no contexto da tecnologia alvo, e *(ii)* na estrutura e características da tecnologia alvo para a definição dos seus operadores de mutação. Partindo desse princípio, diferentes trabalhos, como os (DELAMARO; MALDONADO; MATHUR, 2001) e (FERRARI; MALDONADO; RASHID, 2008) citados anteriormente, por exemplo, iniciaram suas pesquisas a partir de um estudo sobre defeitos nos contextos alvo e utilizaram os resultados desse estudo para, então, definir operadores de mutação. Nosso trabalho aplica essa mesma estratégia para projetar operadores de mutação no contexto de programas de processamento de Big Data. O desenvolvimento e resultados do nosso trabalho são apresentados nos capítulos seguintes.

3 Sistemas de Processamento de Big Data

Este capítulo apresenta sistemas de processamento paralelo e distribuído em *cluster* (grupos) de computadores que são utilizados para o processamento de grandes volumes de dados (*Big Data*). Os sistemas apresentados são o *Apache Hadoop* (HADOOP, 2019), *Dryad* (ISARD et al., 2007) e *DryadLINQ* (YU et al., 2008), *Nephele/PACTs* (BATTRÉ et al., 2010), *FlumeJava* (CHAMBERS et al., 2010) e o *Apache Spark* (ZAHARIA et al., 2010). Nosso objetivo é apresentar as principais características desses sistemas e discutir suas semelhanças e diferenças em relação ao modelo de programação, fluxo de dados, representação dos dados e interface de programação.

Este capítulo está organizado da seguinte forma: na Seção 3.1, apresentamos o sistema Apache Hadoop e o modelo MapReduce. A Seção 3.2 apresenta o sistema Dryad e o DryadLINQ. O sistema Nephele e o seu modelo PACTs é apresentado na Seção 3.3. Na Seção 3.4, apresentamos o FlumeJava e o Apache Beam. A Seção 3.5 apresenta o Apache Spark. Por último, a Seção 3.6 apresenta uma discussão sobre as semelhanças entre os sistemas de processamento de Big Data apresentados.

3.1 Apache Hadoop

O *Apache Hadoop* é um sistema de código aberto para armazenamento e processamento de dados distribuídos em *cluster* de computadores (HADOOP, 2019). O Hadoop implementa o modelo *MapReduce* (DEAN; GHEMAWAT, 2004), um modelo para programação paralela em arquiteturas do tipo *cluster* inspirado nas operações *map* e *reduce* da programação funcional. Este modelo, proposto pela *Google*, ganhou destaque por ter sido um dos primeiros a prover um modelo simplificado e escalável para o processamento paralelo e distribuído em larga escala, o tornando ideal para o processamento de grandes volumes de dados (*Big Data*). Além de implementar o MapReduce, o Hadoop também fornece o sistema de arquivos distribuídos *Hadoop Distributed File System* (HDFS) (SHVACHKO et al., 2010), que é responsável por armazenar e gerenciar os dados distribuídos no cluster de computadores. Com o Hadoop, detalhes complexos do ambiente distribuído como paralelização, distribuição dos dados, tolerância a falhas e balanceamento de carga podem ser abstraídos pelo desenvolvedor, de modo que este pode focar nos aspectos algorítmicos do processamento de dados de seu programa. O Apache Hadoop se tornou uma infraestrutura para uma série de sistemas de Big Data, como o *Hive* (THUSOO et al., 2009) e o *Pig* (OLSTON et al., 2008), entre outros. A seguir, apresentamos mais detalhes sobre o modelo MapReduce que é implementado pelo Apache Hadoop.

O modelo de programação MapReduce consiste em duas etapas: mapeamento

(*Map*) e redução (*Reduce*). Em ambas, o desenvolvedor escreve funções que recebem pares chave/valor como entrada e produzem pares chave/valor como saída. A função *Map* recebe como entrada um par chave/valor e produz como saída zero ou mais pares chave/valor. Em seguida, esses pares chave/valor são agrupados de acordo com a chave e redistribuídos para serem passados para a etapa de Redução. A função *Reduce* recebe como entrada um par chave/valor em que o valor é uma sequência contendo todos os valores que foram associados com esta chave. Então, a função processa esse grupo de valores e produz zero ou mais pares chave/valor como saída. A Figura 3.1 mostra uma visão geral do processo de execução de um programa MapReduce no Hadoop, as suas principais etapas são descritas a seguir:

Particionamento: nesta etapa, o conjunto de dados que será processado é dividido em um número fixo de diferentes partes (chamadas de *partições*) e distribuído entre os diferentes computadores (nós) do cluster. No Hadoop, este particionamento é gerenciado pelo HDFS. Os resultados das etapas seguintes também são armazenados e gerenciados pelo HDFS;

Mapeamento: nesta etapa, cada partição é processada por um processo que executa a função Map definida pelo desenvolvedor. Este processamento respeita a localização das partições, de forma que cada partição é processada no nó em que ela está alocada no cluster. O resultado desta etapa é um conjunto de pares chave/valor gerados pelas funções Map;

Redistribuição (*Shuffling*): esta etapa reorganiza os pares chave/valor gerados pela etapa de Mapeamento. Uma vez que cada processo Reduce recebe como entrada uma sequência de valores que compartilham a mesma chave, é necessário que todos os valores que possuam uma chave em comum estejam em um mesmo nó no cluster. Dessa forma, esta etapa reorganiza os dados no cluster ao ordenar os pares pela chave e redistribuir os valores para agrupá-los em um mesmo nó no cluster;

Redução: nesta etapa, os valores que possuem uma chave em comum são processados em um mesmo processo que executa a função Reduce definida pelo desenvolvedor. A função Reduce processa esse grupo de valores e produz um único par chave/valor como saída.

A Figura 3.2 apresenta o exemplo de uma aplicação MapReduce no Hadoop. Essa aplicação conta o número de ocorrências de cada palavra em um conjunto de dados. A implementação da etapa de Mapeamento, a partir do método *map*, pode ser vista entre as linhas 4 e 12. O método *map* recebe como entrada uma chave e um valor, em que o valor consiste em uma linha de texto do conjunto de dados que está sendo processado (linha 5). Esta linha de texto é separada em palavras na linha 6. Então, para cada palavra, é

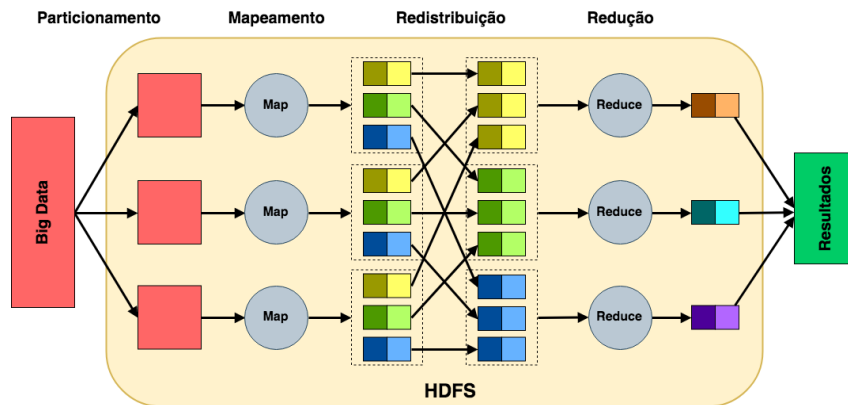


Figura 3.1 – Visão geral do processo de execução do modelos *MapReduce* no Apache Hadoop.

emitida um elemento chave/valor em que a chave é a palavra e o valor o número inteiro 1 (linhas 7 à 11). A implementação da etapa de Redução, a partir do método *reduce*, pode ser vista entre as linhas 16 e 24. O método *reduce* recebe como entrada uma chave, que corresponde a uma palavra, e um conjunto de valores que correspondem a todos os elementos que foram associados com aquela mesma chave. Esses valores são somados entre as linhas 18 e 21. Essa soma, que corresponde ao número de ocorrências da palavra, é emitida junto com a palavra na linha 23. Todo o processo de execução, assim como a etapa de Redistribuição que ocorre entre o Mapeamento e a Redução, é coordenada pelo sistema Hadoop, que lê e escreve as entradas, saídas e resultados intermediários no HDFS.

O sistema Hadoop e o modelo MapReduce possuem como vantagens *simplicidade*, uma vez que abstrai para o desenvolvedor complexidades da programação paralela e distribuída; *flexibilidade*, uma vez que permite processar dados em qualquer formato, sem dependência de algum modelo ou esquema; *tolerância a falhas*, uma vez que falhas são comuns em ambientes de cluster de computadores; e *escalabilidade*, uma vez que suporta clusters com centenas ou milhares de computadores (LEE et al., 2012).

Apesar das vantagens, estes também possuem uma série de limitações, como destacado em (KALAVRI; VLASSOV, 2013):

- *Desempenho*: mesmo oferecendo uma escalabilidade que permite que centenas de computadores sejam utilizados no processamento, programas MapReduce perdem desempenho de execução devido ao tempo gasto pelo sistema com processos de inicialização, agendamento, coordenação e monitoramento de suas etapas. Além disso, todas as etapas do MapReduce demandam leitura e escrita de entradas e saídas no HDFS, algo que contribui consideravelmente para o tempo de execução;
- *Modelo*: mesmo oferecendo um modelo simplificado de programação, desenvolver programas MapReduce exige grandes habilidades pelo desenvolvedor. Seu modelo rígido, que obriga o programa ser escrito em etapas fixas de Map e Reduce, faz

```
1 public class WordCount {
2
3     public static class TokenizerMapper extends Mapper<Object, Text, Text,
4         IntWritable>{
5         public void map(Object key, Text value, Context context) throws
6             IOException, InterruptedException {
7             String line = value.toString();
8             String[] words = line.split(" ");
9             for(String w : words) {
10                 Text word = new Text(w);
11                 IntWritable one = new IntWritable(1);
12                 context.write(word, one);
13             }
14         }
15     }
16
17     public static class IntSumReducer extends Reducer<Text, IntWritable, Text,
18         IntWritable> {
19         public void reduce(Text key, Iterable<IntWritable> values, Context
20             context) throws IOException, InterruptedException {
21             Text word = key;
22             int sum = 0;
23             for (IntWritable val : values) {
24                 sum += val.get();
25             }
26             IntWritable count = new IntWritable(sum);
27             context.write(word, count);
28         }
29     }
30 }
```

Figura 3.2 – Exemplo de aplicação de contagem de palavras no Hadoop MapReduce.

com que desenvolver operações comuns, como uma junção por exemplo, não seja uma tarefa trivial uma vez que nem todas as operações são facilmente expressadas em Map e Reduce. Por esse motivo, análises mais complexas precisam ser escritas como uma sequência de programas MapReduce em que os resultados de um são utilizados como entradas para os seguintes. Isso faz com que o modelo MapReduce não seja adequado para uma série de aplicações, como algoritmos de aprendizagem de máquina (*machine learning*) e processamento de grafos que podem exigir várias iterações, por exemplo;

- *Configuração*: mesmo abstraindo processos de distribuição e paralelismo, preparar uma infraestrutura para execução de programas MapReduce com o Hadoop exige a configuração de uma série de parâmetros que estão ligados diretamente ao desempenho de execução. Esses parâmetros incluem o nível de paralelismo, número e tamanho das partições e o fator de replicação das partições. Configurar de forma adequada o cluster exige uma série de conhecimentos sobre a infraestrutura, como saber a quantidade de recursos disponíveis, e sobre as características da carga de trabalho, fazendo com que escolhas erradas tenham uma consequência direta no

desempenho.

Os problemas e limitações do MapReduce vem sendo pesquisados de maneira ativa desde a sua publicação em 2004 (LEE et al., 2012). Essas pesquisas impulsionaram a criação de novos modelos de programação e sistemas voltados para Big Data. Alguns projetos optaram por estender o modelo MapReduce para permitir seu uso em programas iterativos e de fluxo contínuo de dados, como os projetos *HaLoop* (BU et al., 2010) e *DEDUCE* (KUMAR et al., 2010). Outros projetos propuseram novos modelos de programação e sistemas que abordaram de forma direta as limitações do modelo MapReduce e Hadoop, como o *Dryad* (ISARD et al., 2007) e *DryadLINQ* (YU et al., 2008), Nephel/PACTs (BATTRÉ et al., 2010), *FlumeJava* (CHAMBERS et al., 2010) e *Apache Spark* (ZAHARIA et al., 2010). Esses sistemas serão apresentados a seguir.

3.2 Dryad e DryadLINQ

O *Dryad* (ISARD et al., 2007) é um sistema e modelo para programação distribuída desenvolvido pela *Microsoft*. O *DryadLINQ* (YU et al., 2008) é uma interface de programação baseada no *LINQ* (*Language INtegrated Query*) (MEIJER; BECKMAN; BIERMAN, 2006) que simplifica o desenvolvimento de aplicações para o Dryad.

3.2.1 Dryad

O *Dryad* (ISARD et al., 2007) é um sistema distribuído de propósito geral para execução de aplicações de processamento paralelo de dados. No Dryad, aplicações são representadas como um *Grafo Acíclico Orientado* (*Directed Acyclic Graph* ou DAG, em inglês) em que os vértices são programas e arestas são canais de comunicação. Os vértices são automaticamente mapeados para os computadores no cluster em tempo de execução e os canais de comunicação são utilizados para transportar os dados de entrada e saída dos vértices. O Dryad suporta três tipos de canais de comunicação: *i*) em memória, *ii*) em rede usando o protocolo TCP e *iii*) em arquivos temporários no sistema de arquivos.

O Dryad foi projetado para oferecer um modelo de fluxo de dados flexível que não restringe seus vértices a consumirem e produzirem um único conjunto de dados, como ocorre com o modelo MapReduce. Dessa forma, os vértices podem receber e produzir um número arbitrário de entradas e saídas. Esse modelo faz com que o Dryad perca em simplicidade com relação ao MapReduce, mas, em contrapartida, faz ele ganhar em expressividade ao oferecer um maior controle do fluxo de dados para o desenvolvedor e possibilitar a aplicação de um maior número de operações que não são limitadas apenas a operações do tipo *Map* e *Reduce*.

No Dryad, uma aplicação é representada como um grafo $G = \langle V_G, E_G, I_G, O_G \rangle$ em que V_G é o conjunto de vértices, E_G é um conjunto de arestas direcionadas, e I_G e O_G que são subconjuntos de V_G e representam os conjuntos de vértices de entrada e saída, respectivamente. Ao se criar um vértice v , esse pode ser considerado como um grafo único em que $G = \langle \{v\}, \emptyset, \{v\}, \{v\} \rangle$. A DAG completa da aplicação é definida a partir da composição de vértices e subgrafos com operadores que permitem criar várias cópias de um único vértice ($\wedge$), definir arestas entre dois grafos ($\geq$ e $\gg$) e unir dois grafos ($\parallel$).

A execução de uma aplicação é coordenada pelo *job manager*, principal componente do sistema Dryad. Ele é responsável por *i*) instanciar a DAG da aplicação; *ii*) agendar e coordenar a execução dos vértices nos computadores do cluster; *iii*) prover tolerância a falhas ao reexecutar tarefas que falharam ou estão lentas; *iv*) monitorar a execução e coletar estatísticas; e *v*) fazer transformações na DAG dinamicamente para fazer otimizações durante a execução.

3.2.2 DryadLINQ

O *DryadLINQ* (YU et al., 2008) oferece uma interface de alto nível para o desenvolvimento de aplicações de processamento de dados em larga escala. O DryadLINQ expande o *LINQ* (*Language INtegrated Query*) (MEIJER; BECKMAN; BIERMAN, 2006) para o ambiente de programação distribuída e paralela do Dryad. O LINQ é uma extensão de linguagem e API para consulta e manipulação de coleções de dados da plataforma .NET. Ele fornece uma interface híbrida de programação declarativa e imperativa, permitindo que programas sejam escritos em uma linguagem similar ao SQL ou através da chamada de métodos e passagem de funções.

As principais abstrações no LINQ são as interfaces *IEnumerable<T>*, que representa um conjunto de dados abstrato do tipo T , e *IQueryable<T>*, que é subtipo de *IEnumerable* e representa expressões construídas a partir da combinação de conjuntos de dados e operadores LINQ. Essas duas interfaces permitem que programas sejam escritos sem que o desenvolvedor precise saber como as estruturas foram concretamente implementadas. Além disso, as expressões LINQ são avaliadas de maneira tardia (*lazy evaluation*), de forma que só são computadas quando necessário.

O LINQ oferece um extenso conjunto de operadores que permitem fazer mapeamentos, filtrações, agregações e ordenações em coleções de dados, além de outros tipos de operações. O conjunto completo de operações do LINQ pode ser visto em (LINQ, 2017), algumas das principais operações são descritas a seguir:

Select mapeia todos os elementos da coleção para uma nova forma com base em uma função de transformação passada como parâmetro.

SelectMany opera de forma semelhante a *Select*, mas ao invés da função mapear cada elemento para um novo valor, ela mapeia cada elemento para uma nova coleção de valores e o resultado da operação é a concatenação de todas as coleções geradas pela função e uma única coleção. Essa operação se assemelha à etapa de *map* do modelo MapReduce.

Where filtra os elementos de uma coleção com base em uma função de predicado passada como parâmetro.

GroupBy agrupa os elementos de uma coleção de acordo com uma função seletora de chave passada como parâmetro, cada grupo na coleção resultante possui uma chave em comum.

Aggregate agrega os elementos de uma coleção em um único valor com base em funções de agregação passadas como parâmetro. Essa operação representa uma agregação personalizada da coleção, mas o LINQ também oferece implementações de agregações comuns, como a soma e contagem dos elementos da coleção por exemplo.

Distinct retorna os elementos distintos de uma coleção.

Union produz a união entre duas coleções de forma que a coleção resultante possui todos os elementos que estavam presentes nas duas coleções sem repetição. Essa operação possui o mesmo comportamento da operação de união de conjuntos matemáticos. Outras operações de conjuntos matemáticos como a interseção e diferença entre conjuntos também estão presentes no LINQ.

Join realiza a junção entre duas coleções com base em uma função seletora de chave passada como parâmetro. A coleção resultante possui pares com elementos das duas coleções que compartilham uma mesma chave.

OrderBy ordena os elementos de uma coleção com base em uma função seletora de chave passada como parâmetro. Por padrão, a coleção é ordenada de forma ascendente, mas também é possível ordenar em forma descendente.

O *DryadLINQ* (YU et al., 2008) compila programas LINQ para computações distribuídas na infraestrutura do Dryad. Nele, os conjuntos de dados ainda são representados de forma abstrata como uma coleção de objetos *.NET*, mas os seus elementos estão distribuídos pelos computadores de um cluster. Esse modelo de representação do conjunto de dados é apresentado na Figura 3.3. Expressões LINQ são otimizadas pelo sistema DryadLINQ e convertidas em DAGs Dryad de forma eficiente. Ao unir o modelo expressivo de manipulação de dados do LINQ com o modelo de execução distribuída do Dryad, o DryadLINQ permite que desenvolvedores escrevam aplicações de processamento

de dados de forma mais simplificada, sem ser necessário se preocupar com a complexidade de ambientes distribuídos.

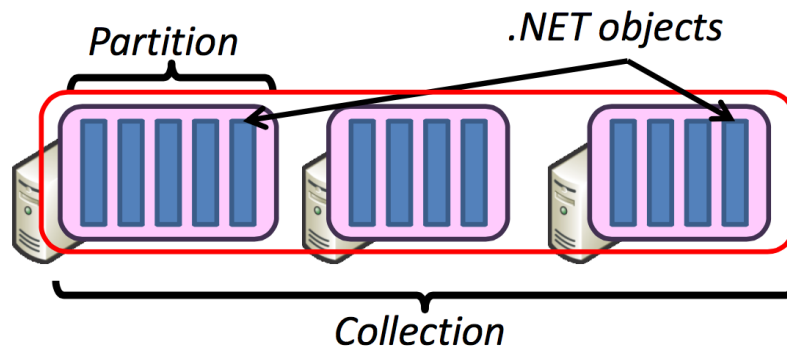


Figura 3.3 – Modelo de dados do *DryadLINQ*.

Fonte: (YU et al., 2008)

As coleções no DryadLINQ são representadas por objetos do tipo *DryadTable* $\langle T \rangle$, que é um subtipo de *IQueryable* $\langle T \rangle$ do LINQ. Objetos *DryadTable* podem ser criados a partir de diferentes fontes de dados, como sistemas de arquivos distribuídos ou tabelas SQL, e podem conter metadados, como esquemas de tabelas ou informações sobre particionamentos. Essas informações podem ser utilizadas pelo otimizador do DryadLINQ para fazer otimizações na execução. O particionamento dos dados no DryadLINQ é feito de maneira transparente pelo sistema, mas ele também oferece a possibilidade de modificação pelo desenvolvedor a partir de operações que re-particionam a coleção com base em *hash* (*HashPartition*) ou intervalos (*RangePartition*). Essas operações também podem ser utilizadas para modificar escolhas de otimização do plano de execução feito pelo DryadLINQ.

Entradas e saídas em um programas DryadLINQ são definidas pelas operações *GetTable* $\langle T \rangle$ e *ToDryadTable* $\langle T \rangle$, que recebem o endereço de onde os dados estão sendo lidos ou para onde serão salvos como entrada. Devido ao processamento em paralelo dos dados, o DryadLINQ impõe que todas as funções passadas como parâmetro nas expressões sejam livres de efeito colateral (*side-effect free*). Essas funções podem referenciar objetos compartilhados, mas o DryadLINQ não faz nenhuma verificação para garantir que estes não estão sendo modificados e não faz garantias sobre o resultado das expressões quando isso ocorre. Além das operações já existentes no LINQ, o DryadLINQ acrescenta mais duas operações que são descritas a seguir:

Apply mapeia os elementos de uma coleção para uma nova forma com base em uma função de transformação passada como parâmetro. Entretanto, ao contrário da operação *Select* do LINQ que aplica a função em cada elemento da coleção individualmente, a operação *Apply* aplica a função em toda a coleção de dados que é acessada a partir de um iterador e o seu resultado é uma coleção completa.

Fork opera de forma semelhante à *Apply*, mas ao invés de gerar uma única coleção a partir de outra, ela pode gerar várias coleções de dados como saída.

A Figura 3.4 apresenta o exemplo de uma aplicação no DryadLINQ que conta a frequência de palavras em um conjunto de dados. Inicialmente, é feita uma separação dos textos do conjunto de dados de entrada em palavras (linha 2). Em seguida, as palavras são agrupadas de modo que a chave é a própria palavra (linha 3). Os valores associados com a chave também são a própria palavra, de forma que a quantidade de vezes que a palavra aparece no texto, é a quantidade de elementos com mesmo valor no grupo. Dessa forma, a frequência de cada palavra é obtida a partir da quantidade de elementos no grupo (linha 4). De maneira geral, a lógica dessa aplicação no DryadLINQ segue a mesma lógica da implementação no Hadoop MapReduce, apresentada na Figura 3.2. Entretanto, esta segue um fluxo de dados em que a lógica da aplicação é determinada por sucessivas aplicações de operações no conjunto de dados, enquanto que no Hadoop MapReduce cada etapa é implementada com um fluxo de controle.

```
1 var input = DryadLinq.GetTable<Doc>("file://path")
2 var words = input.SelectMany(docs => docs.Line.Split(' '));
3 var groups = words.GroupBy(word => word);
4 var counts = groups.Select(group => new Pair(group.Key, group.Count()));
5 counts.ToDryadTable("output.txt");
```

Figura 3.4 – Exemplo de aplicação de contagem de palavras no DryadLINQ.

3.2.3 Execução

O fluxo de execução de um programa DryadLINQ é apresentada na Figura 3.5. Seus passos são descritos a seguir:

- (1) Um programa que cria um objeto com expressões DryadLINQ é executado. Devido ao seu modelo de avaliação tardia, a execução das expressões não são iniciadas de fato.
- (2) A invocação da operação *ToDryadTable* desencadeia a execução em paralelo das expressões.
- (3) O DryadLINQ compila as expressões LINQ em um plano de execução distribuída no Dryad. Nesse processo, o compilador decompõe as expressões em subexpressões, de forma que cada uma vai ser executada em um vértice separado na DAG, gera código e dados estáticos para vértices remotos e gera códigos de serialização para os tipos de dados requisitados. Nesta etapa o DryadLINQ também faz otimizações estáticas no plano de execução. Essas otimizações incluem unir operações em um

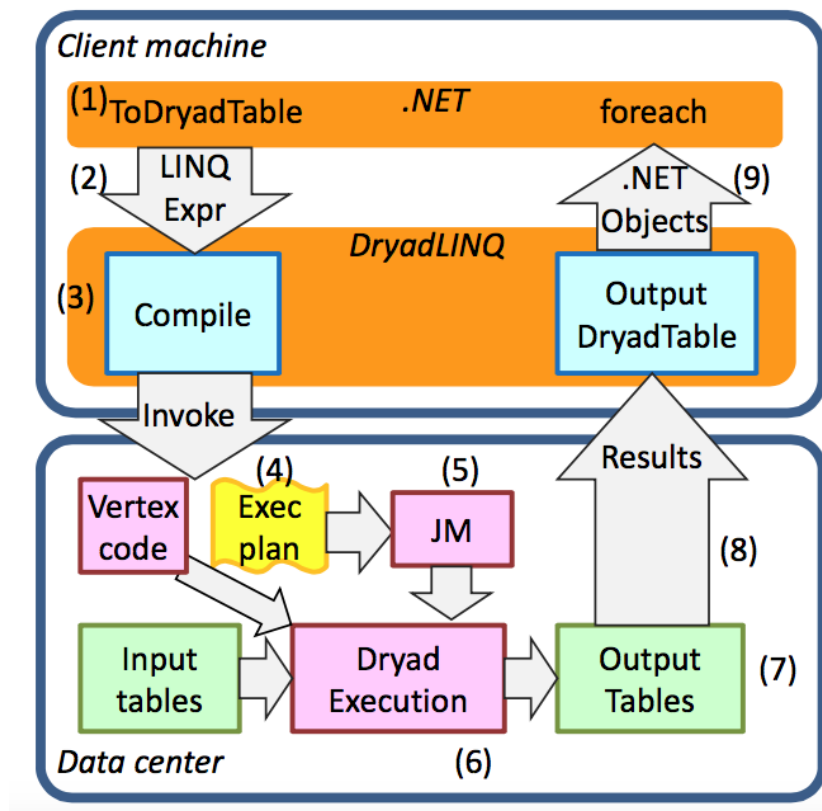


Figura 3.5 – Visão geral da execução de um programa *DryadLINQ*.

Fonte: (YU et al., 2008)

mesmo processo (*pipelining*), remover redundâncias em etapas de particionamento, antecipar agregações para reduzir a quantidade de dados trafegados na rede (*eager aggregation*) e reduzir o tempo gasto com leitura e escrita de entradas e saídas ao utilizar canais de comunicação em memória ou rede sempre que possível ao invés de arquivos.

- (4) O sistema invoca um *job manager* (JM) específico do DryadLINQ para coordenar a execução e fazer otimizações dinâmicas.
- (5) O JM cria um grafo de execução (DAG) com base no plano de execução criado no passo (3).
- (6) O Dryad executa o código de cada um dos vértices contidos na DAG seguindo a ordem indicada pelo JM.
- (7) Quando a execução de todo o grafo é finalizada com sucesso, o Dryad escreve os resultados nas tabelas de saída (*Output Tables*).
- (8) O processo do JM é finalizado e o controle da aplicação é retornado para o DryadLINQ. O DryadLINQ cria objetos locais do tipo *DryadTable* que referenciam as

tabelas de saída. Esses objetos podem ser utilizados em novas expressões ou ter seus elementos acessados no programa do usuário.

- (9) O controle retorna para a aplicação do usuário que pode acessar o conteúdo da *DryadTable* através de uma interface de iterador.
- (10) A aplicação pode gerar novas expressões DryadLINQ a serem executadas com a repetição dos passos (2)–(9).

3.3 Nephele/PACTs

O Nephele/PACTs (BATTRÉ et al., 2010) é um sistema de processamento de dados em larga escala que fornece *Contratos de Paralelização* (PACTs) como modelo de programação e estes são automaticamente otimizados para serem executados no *Nephele* (WARNEKE; KAO, 2009), um motor de execução de sistemas distribuídos em nuvem. A Figura 3.6 descreve o processo de execução do sistema, nele um programa escrito no modelo PACTs é compilado para um programa de fluxo de dados em grafo no modelo do Nephele e, em seguida, este é executado em um ambiente de cluster em nuvem. O Nephele/PACTs forma a base do projeto Stratosphere (ALEXANDROV et al., 2014), uma plataforma completa para análise de Big Data.

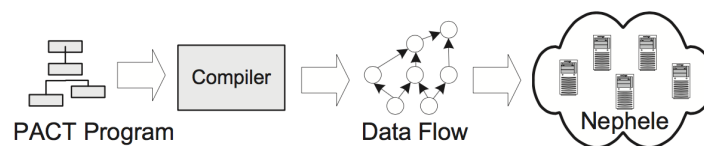


Figura 3.6 – Processo de compilação e execução no Nephele/PACTs.

Fonte: (BATTRÉ et al., 2010)

3.3.1 Nephele

O sistema *Nephele* (WARNEKE; KAO, 2009) é um motor de execução distribuída para programas implementados como DAGs. O Nephele trabalha de forma semelhante ao Dryad (ISARD et al., 2007). Nele, as arestas representam canais de comunicação que transferem dados entre os subprogramas. Os vértices da DAG são programas sequenciais executáveis que processam dados dos canais de entrada e escrevem seus resultados nos canais de saída.

O Nephele se encarrega do agendamento de tarefas e configuração dos canais de comunicação, além de oferecer um mecanismo de tolerância a falhas que ajuda a minimizar o impacto de falhas de Hardware durante a execução. O Nephele suporta alocamento dinâmico de recursos de Hardware em ambientes de nuvem, como o *Amazon EC2*, por

exemplo. Isso faz com que o Nephele se destaque em relação ao Dryad que é projetado para fazer alocação estática dos recursos de Hardware. Entretanto, o Nephele não suporta otimizações em tempo de execução como o Dryad.

Assim como o Dryad, o Nephele possui três tipos de canais de comunicação: *i)* em memória (*in memory*), *ii)* rede (*network*) e *iii)* arquivo (*file*). Os canais de rede e em memória permitem uma execução com baixa latência de forma que uma tarefa pode consumir os dados de saída de outra de maneira rápida. Já os canais de comunicação de arquivos salvam todo o conteúdo de saída de uma tarefa em arquivos temporários antes de passá-los para a tarefa seguinte, fazendo com que estes também funcionem como pontos de checagem que podem ajudar nos casos em que ocorre alguma falha de execução. Para persistência e armazenamento de dados o Nephele utiliza sistemas de arquivos distribuídos como o HDFS.

3.3.2 PACTs

O modelo de programação *PACT* é representado como um grafo de fluxo de dados que contem nós de entrada, nós de saída e nós de processamento que são chamados *Contratos de Paralelização* (PACTs) (BATTRÉ et al., 2010). Este pode ser visto como uma generalização do modelo de programação MapReduce (DEAN; GHEMAWAT, 2004). Neste modelo, programas são implementados através do fornecimento de um código sequencial que representa alguma tarefa específica (chamada de *função do usuário*) para algum PACT e anexando este a um fluxo de dados. Um PACT opera sobre dados do tipo chave/valor e define propriedade da entrada e saída dos dados.

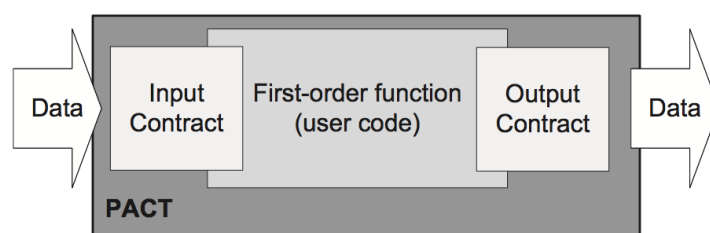


Figura 3.7 – Componentes de um PACT.

Fonte: (BATTRÉ et al., 2010)

A Figura 3.7 mostra os componentes de um PACT. Podemos ver que ele é composto de um *Contrato de Entrada* (*Input Contract*), uma função do usuário (*First-order function*) e um *Contrato de Saída* (*Output Contract*), sendo que este último é opcional. O contrato de entrada define como a função do usuário é executada em paralelo e o contrato de saída define propriedades sobre os valores retornados pela função do usuário que podem ajudar na otimização da execução do programa.

No processamento paralelo de dados, não é necessário ter conhecimento de todo o contexto do conjunto de dados para processar um elemento. Dessa forma, subconjuntos do conjunto de dados conseguem ser processados de maneira independente um do outro. No modelo PACTs, esses subconjuntos são chamados de *Unidades de Paralelização* (UP). A quantidade de UPs é o que determina o nível máximo de paralelização de uma aplicação uma vez que cada UP pode ser processada de forma independente. Nesse contexto, os Contratos de Entrada são funções de segunda ordem que definem como os dados de entrada são mapeados em UPs e que fazem a chamada da função do usuário para cada UP. Podem receber como entrada um único conjunto de dados ou múltiplos conjuntos de dados. O modelo possui cinco contratos de entrada padrão, os contratos *Map* e *Reduce*, que recebem um único conjunto de dados de entrada, e os contratos *Cross*, *CoGroup* e *Match*, que recebem múltiplos conjuntos de entrada. Esses contratos são descritos a seguir:

Map opera de forma igual à etapa de *map* no modelo MapReduce. Este contrato considera cada par chave/valor como uma UP e, conseqüentemente, faz a chamada da função do usuário independentemente pra cada elemento no conjunto de dados.

Reduce faz com que cada par chave/valor que possui a mesma chave sejam atribuídas à mesma UP, de forma que os elementos no conjunto de dados são particionados pela chave. Em seguida, cada UP é processada por uma instância da função do usuário, fazendo com que este contrato tenha o mesmo comportamento da etapa *reduce* no modelo MapReduce.

Cross é definido como o produto Cartesiano sobre os elementos dos seus conjuntos de entrada. Após, a função do usuário é aplicada em cada elemento resultante do produto Cartesiano.

CoGroup particiona os elementos chave/valor de seus conjuntos de entrada de acordo com a chave. Em seguida, os elementos que possuem a mesma chave são atribuídos a uma mesma UP e processada por uma mesma instância da função do usuário. Este contrato opera de forma semelhante ao *Reduce*, entretanto aceita múltiplos conjuntos de entrada.

Match de forma semelhante ao CoGroup, este contrato particiona os conjuntos de dados de acordo com a chave. Entretanto, ao invés de criar uma UP para todos os valores de uma mesma chave, este cria uma UP para cada combinação de dois elementos com a mesma chave e cada combinação é processada por uma instância da função do usuário. Neste contrato, se algum elemento chave/valor de um dos conjuntos de dados de entrada não tiver algum elemento com a chave correspondente no outro conjunto, este elemento não é processado pela função do usuário, fazendo com que o *Match* opere como uma junção interna (*inner join*).

Os cinco contratos são representados na Figura 3.8. Na figura, as formas desenhadas com linhas tracejadas representam as unidades de paralelização (UPs) que cada contrato define com base nos conjuntos de pares chave/valor de entrada. As setas indicam como os pares nos conjuntos de entrada são mapeados para UPs. Em seguida, cada UP é processada de forma independente por uma instância da função do usuário.

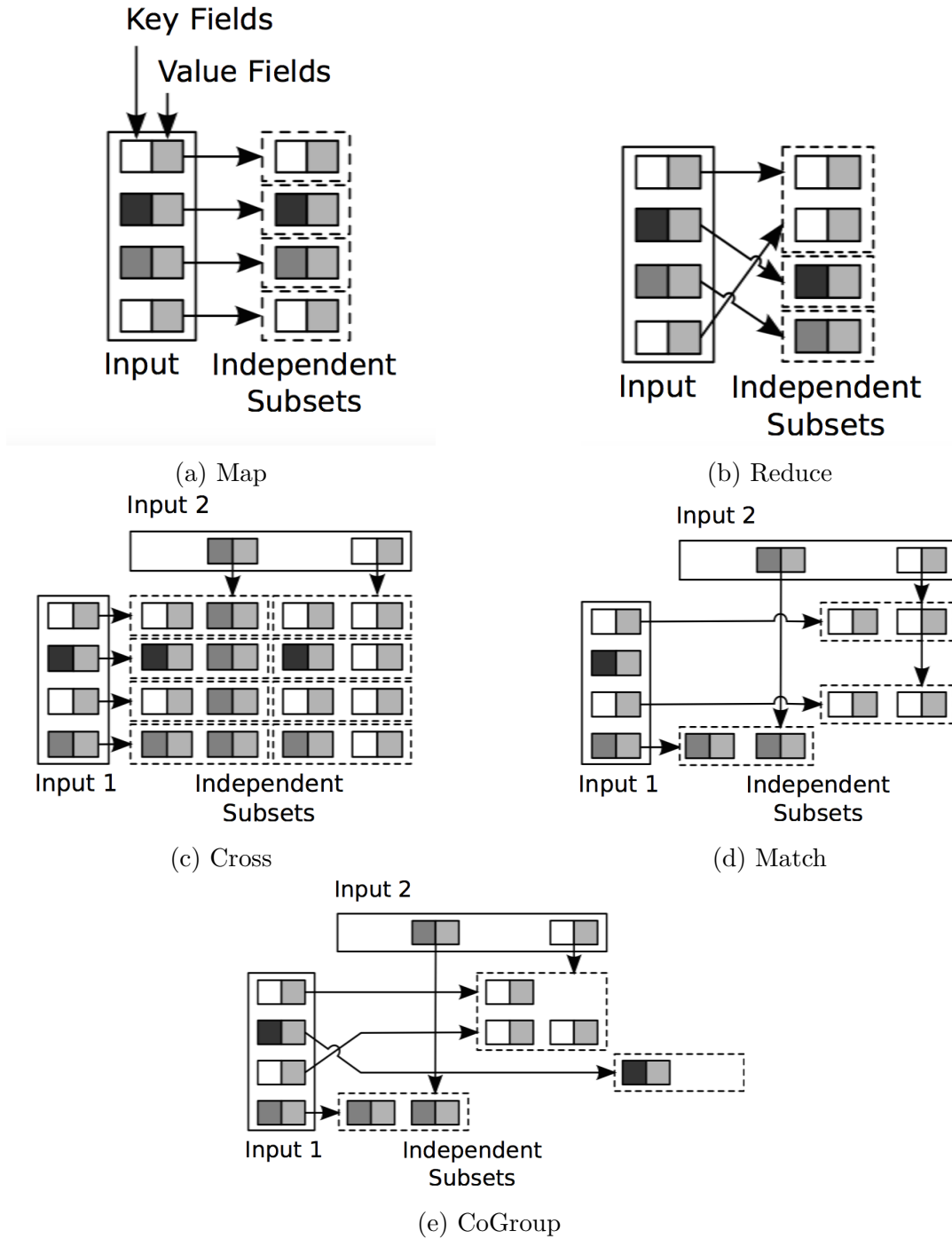


Figura 3.8 – Representação dos cinco Contratos de Paralelização de Entrada do Nephle/PACTs.

Fonte: (HUESKE et al., 2012)

Os contratos de saída são componentes opcionais dos PACTs que especificam

propriedades sobre os elementos que são produzidos pelas funções do usuário. Essas propriedades são definidas em termos da chave dos pares chave/valor de saída e podem ter relação com a chave dos pares de entrada. As propriedades não são forçadas pelo sistema, elas devem ser garantidas pela função do usuário. Logo, erros ao definir o contrato de saída para a função do usuário podem acarretar em comportamento errado ou ineficiente do programa. Esses contratos de saída são utilizados pelo sistema para inferir quando o particionamento e a ordenação das chaves deve ser mantida e, assim, fazer otimizações na execução do programa. Os quatro contratos de saída são definidos a seguir:

Same-Key impõe que cada par chave/valor gerado pela função do usuário deve possuir a mesma chave dos valores que o geraram. Dessa forma, o contrato faz com que as propriedades de particionamento e ordem das chaves sejam preservadas.

Super-Key impõe que cada par chave/valor que é gerado pela função do usuário possui uma super chave dos pares chave/valor que o geraram. Dessa forma, a função preserva o particionamento e a ordem parcial das chaves.

Unique-Key impõe que cada par chave/valor produzido pela função do usuário deve possuir uma chave única em todo o conjunto de dados. Os pares gerados são particionados de acordo com a chave.

Partitioned-by-Key impõe que os pares chave/valor gerados pela função do usuário devem manter o mesmo particionamento das chaves. Este contrato trabalha de forma similar à *Super-Key* ao fazer com que o particionamento pela chave seja mantido, mas não impõe nenhuma ordem dentro das partições.

A Figura 3.9 apresenta o exemplo da aplicação de contagem de palavras implementada no PACTs. A implementação de um PACT segue a mesma forma de implementação de uma etapa no Hadoop MapReduce, de modo que cada contrato de entrada determina a interface do método a ser implementado pelo desenvolvedor e sua implementação segue uma lógica de fluxo de controle. Dessa forma, a aplicação pode ser implementada da mesma forma em que foi implementada no Hadoop MapReduce com a implementação de um contrato *Map* e um contrato *Reduce*. Assim sendo, a implementação do *Map* (linhas 3 à 13) separa um texto em palavras e emite um par chave/valor com cada palavra e o número inteiro 1, e a implementação do contrato *Reduce* (linhas 15 à 26) faz a soma dos valores associados com a chave, possuindo a mesma lógica de implementação do exemplo na Figura 3.2. O que diferencia ambas as aplicações é a definição do plano de execução, ou fluxo de operações, que deve ser determinado de forma explícita no PACTs (linhas 28 à 24). Enquanto o Hadoop MapReduce possui um modelo rígido com duas operações e ordem de execução fixas, o PACTs é flexível ao permitir que mais operações sejam feitas seguindo a ordem definida pelo desenvolvedor.

```

1  public class WordCount implements PlanAssembler,
    PlanAssemblerDescription {
2
3      public static class TokenizeLine extends MapStub {
4          public void map(PactRecord record, Collector<PactRecord> collector)
5          {
6              String line = record.getField(0, PactString.class).getValue();
7              String[] words = line.split(" ");
8              for (String w : words) {
9                  PactString word = new PactString(w);
10                 PactInteger one = new PactInteger(1);
11                 collector.collect(new PactRecord(word, one));
12             }
13         }
14
15         public static class CountWords extends ReduceStub {
16             public void reduce(Iterator<PactRecord> records, Collector<
17                 PactRecord> collector) throws Exception {
18                 PactString word = null;
19                 int sum = 0;
20                 for (PactRecord val : records) {
21                     word = word != null ? word : val.getField(0, PactString.class);
22                     sum += val.getField(1, PactInteger.class).getValue();
23                 }
24                 PactInteger count = new PactInteger(sum);
25                 collector.collect(new PactRecord(word, count));
26             }
27
28             public Plan getPlan(String... args) {
29                 FileDataSource source = new FileDataSource(TextInputFormat.class,
30                     dataInput, "Input Lines");
31                 MapContract mapper = MapContract.builder(TokenizeLine.class).input(
32                     source);
33                 ReduceContract reducer = new ReduceContract.Builder(CountWords.class,
34                     PactString.class, 0).input(mapper).build();
35                 FileDataSink out = new FileDataSink(RecordOutputFormat.class, output,
36                     reducer, "Word Counts");
37                 return new Plan(out, "WordCount Example");
38             }
39         }
40     }

```

Figura 3.9 – Exemplo de aplicação de contagem de palavras no Nephele/PACTs.

3.3.3 Compilação e Execução

O processo de execução de um programa PACT pode ser dividido em duas etapas. Na primeira, o programa PACT é passado para um compilador que gera a partir dele uma DAG no modelo do sistema Nephele. Essa DAG possui vértices e arestas e representa de maneira simplificada o fluxo de dados que serão executados em paralelo. Nela, cada vértice possui uma das funções do usuário especificada no programa junto com o código do PACT utilizado. O código do PACT é responsável por fazer o pré-processamento dos dados de entrada de forma a deixá-los na forma da UP definida pelo contrato de entrada. As arestas definem os canais de comunicação que são utilizados para transportar os dados entre as funções do usuário.

Dada as características declarativas dos contratos, o compilador pode utilizar diferentes estratégias de execução que levam em consideração informações como o tamanho dos dados e particionamento para uma melhor otimização. Essas otimizações podem ser a nível de um único PACT, como definir uma melhor estratégia para a criação das UPs que serão processadas pela função do usuário, como também em múltiplos PACTs que estão no programa, como definir que o particionamento e ordenação dos dados devem ser mantidos entre dois PACTs, por exemplo. Após essa etapa de compilação e otimização, a DAG resultante é processada no sistema Nephele.

Na segunda etapa, o sistema Nephele estende a DAG para execução em paralelo. Nesse processo, são criadas múltiplas instâncias dos vértices da DAG. O número de instâncias dos vértices pode variar de acordo com os parâmetros de configuração, como o nível de paralelismo, e as estratégias de cada contrato. Os canais de comunicação associados aos vértices também precisam ser multiplicados. A forma de fazer essa multiplicação também depende das configurações e estratégias definidas nos códigos dos contratos de entrada e saída do vértice. De maneira geral, a estratégia de execução do programa PACT depende do padrão de conexão entre as subtarefas, código do PACT e canais de comunicação.

3.4 FlumeJava e Apache Beam

FlumeJava (CHAMBERS et al., 2010) é um sistema proposto pela *Google* que oferece uma interface de alto nível para o desenvolvimento de aplicações de processamento em paralelo de dados. Foi projetado como uma biblioteca Java centrada no conceito de *coleções paralelas*, que oferecem uma abstração para a forma em que dados distribuídos são representados sem ser necessário ter conhecimento sobre detalhes de baixo nível, como a distribuição física dos dados, particionamento e formatação. Coleções paralelas dão suporte a uma série de *operações paralelas* que podem ser compostas para criar operações mais complexas e desenvolver aplicações que requerem uma sequência de operações (conhecidas como *pipelines*).

O FlumeJava foi criado como uma alternativa ao *MapReduce* (DEAN; GHEMAWAT, 2004), que possui um modelo restrito que faz com que as aplicações tenham que ser desenvolvidas em duas etapas fixas, *map* e *reduce*. Dessa forma, aplicações mais complexas, que exigem um maior número de operações, precisam ser desenvolvidas como uma sequência de várias aplicações *MapReduce*. O FlumeJava simplifica o desenvolvimento de *pipelines* ao permitir que várias operações possam ser feitas em uma única aplicação.

Posteriormente, o FlumeJava foi incorporado ao *Modelo Dataflow* (AKIDAU et al., 2015) da Google. Este modelo unificou os conceitos de processamento de dados do FlumeJava com os conceitos de processamento de fluxo contínuo de dados (*stream processing*) do MillWheel (AKIDAU et al., 2013), também proposto pela Google. O Modelo Dataflow possui uma implementação de código aberto chamada Apache Beam (BEAM, 2016). Este permite que programas de processamento de dados sejam escritos em linguagem Java, Python e Go, além de permitir que os programas sejam executados em diferentes plataformas de execução.

3.4.1 Conceitos

Vamos apresentar os principais conceitos do FlumeJava utilizando a nomenclatura utilizada pelo Apache Beam, uma vez que este é a implementação mais atual deste modelo. O Apache Beam é centrado em três classes e conceitos principais:

Pipeline classe que encapsula todo o processamento dos dados, o que inclui a leitura dos dados de entrada, aplicação de operações e a escrita dos dados de saída. Todas as aplicações Apache Beam devem criar ao menos um objeto Pipeline.

PCollection classe que representa conjuntos de dados distribuídos (*coleções paralelas*) que são processados em pipelines. Geralmente, um pipeline é iniciado com a criação de um objeto PCollection a partir da leitura dos dados de alguma fonte e, a partir deste, novos objetos PCollection são criados através das aplicações de operações. Uma PCollection possui as características de ter seus elementos com um tipo específico, serem imutáveis (após criada, nenhum elemento pode ser adicionado, removido ou alterado), não permitir acesso aleatório a seus elementos e poder ter um número limitado (*batch processing*) ou ilimitado (*stream processing*) de elementos.

PTransform classe que representa uma operação de processamento de dados (*operações paralelas*, que no Apache Beam são chamadas de transformações) no pipeline. Uma transformação pode receber um ou mais objetos PCollection como entrada, realizar algum processamento em paralelo com os dados contidos nessas coleções e produzir zero ou mais objetos do tipo PCollection como saída.

De maneira geral, uma aplicação Apache Beam é iniciada com a criação de um objeto Pipeline e a criação de um ou mais objetos PCollection iniciais, que leem dados de alguma fonte externa. Então, são aplicadas uma série de transformações até uma última que escreve os resultados da execução do pipeline em alguma fonte de dados externa.

3.4.2 Transformações

O Apache Beam oferece seis transformações primitivas e uma série de outras que são construídas a partir dessas. As transformações primitivas são descritas a seguir:

ParDo (*parallelDo* no FlumeJava) é a transformação mais básica do Apache Beam. Ela permite que seja feita alguma computação sobre os elementos de um PCollection de entrada para a geração dos elementos de um PCollection de saída, se assemelhando à etapa de *map* no modelo MapReduce. Esta computação é feita através da aplicação de uma função fornecida pelo desenvolvedor, representada por um objeto *DoFn* que encapsula algum processamento, que recebe um elemento da coleção de entrada e produz zero ou mais elementos da coleção de saída. É a partir desta transformação que operações básicas de mapeamento e filtragem, por exemplo, são implementadas.

GroupByKey é a transformação que permite o processamento de coleções que possuem elementos do tipo chave/valor. Esta operação captura a essência da etapa de *shuffle* do modelo MapReduce, pois recebe como entrada uma PCollection de elementos chave/valor (representados pela classe *KV* que representa uma tupla contendo uma chave e um valor associado a ela), e gera como saída uma PCollection contendo elementos do tipo chave/valor em que o valor é uma coleção contendo todos os valores que eram associados com uma chave única na PCollection inicial.

CoGroupByKey é a transformação que realiza uma junção relacional entre duas coleções do tipo chave/valor que possuem a chave do mesmo tipo.

Combine é a transformação que realiza a combinação (ou agregação) de elementos de uma coleção em um único valor. Pode ser utilizada de duas formas distintas. A primeira, é para combinar todos os elementos de uma PCollection em um único elemento. A segunda forma é para combinar os valores associados a uma chave em um PCollection do tipo chave/valor. A combinação dos elementos é feita através de uma função fornecida pelo desenvolvedor que deve respeitar as propriedades de *associatividade* e *comutatividade* (BEAM, 2016). Essa operação se assemelha às etapas *Combine* e *Reduce* do modelo MapReduce, pois inicialmente os elementos são combinados de forma parcial, respeitando a sua localização física (*combine*), e depois os resultados intermediários são combinados em um único valor (*reduce*).

Flatten é a transformação que recebe como entrada dois ou mais objetos `PCollection` que compartilham elementos do mesmo tipo e retorna um único objeto `PCollection` contendo a união de todos os elementos das coleções de entrada.

Partition é uma transformação que opera de forma contrária à transformação `Flatten`, recebe uma `PCollection` como entrada e divide o seu conteúdo em um número fixo de coleções menores. A divisão dos elementos é feita a partir de uma função fornecida pelo desenvolvedor que determina como os elementos serão distribuídos nas outras coleções.

A partir dessas seis transformações primitivas o Apache Beam fornece uma série de operações derivadas comumente encontradas em outros sistemas, como operações de seleção, agregação e ordenação. Além de permitir que várias transformações sejam compostas para que sejam criadas transformações mais complexas.

A Figura 3.10 apresenta o exemplo da aplicação de contagem de palavras no Apache Beam. A aplicação é iniciada com a criação de um *Pipeline* (linha 2) e de um *PCollection* inicial com a leitura do conjunto de dados de entrada (linha 4). Primeiramente, é aplicada a transformação *ParDo* para separar as linhas do texto em palavras e depois emitir para cada palavra um objeto chave/valor (*KV*) em que a chave é a palavra e o valor o número inteiro 1. Em seguida, é aplicada a transformação *GroupByKey* para agrupar os valores associados a cada chave (linha 16). Por último, é aplicada a transformação *Combine* para somar o grupo de valores associados com cada chave, que resulta na quantidade de vezes em que cada palavra aparece.

3.4.3 Avaliação Diferida, Optimização e Execução

De forma a permitir uma execução otimizada de pipelines, o Apache Beam (assim como o FlumeJava) executa suas operações de maneira tardia (*lazy*) utilizando uma avaliação diferida (*deferred evaluation*). Nesse tipo de avaliação, cada objeto `PCollection` no pipeline é marcado internamente como diferido (ainda não computado) ou materializado (computado). Cada coleção marcada como diferida possui uma referência para a transformação que gerou ela. Por sua vez, cada transformação diferida possui referências para as suas coleções de entrada. O resultado dessas referências entre coleções e transformações não computadas é uma DAG que forma um *plano de execução* do pipeline.

Antes de ser executado, o plano de execução do pipeline é passado por um processo de optimização que inclui a fusão de transformações do tipo `ParDo`, de modo a reduzir o número de etapas na execução, a combinação de operações `ParDo`, `GroupByKey`, `Combine` e `Flatten` em um programa `MapReduce` e a fusão dos programas `MapReduce` gerados. Uma vez optimizado, o plano é então executado seguindo a ordem topológica da DAG. O FlumeJava foi projetado para seguir uma forma de execução em programas `MapReduce`,

```

1 PipelineOptions options = PipelineOptionsFactory.create();
2 Pipeline p = Pipeline.create(options);
3
4 PCollection<String> lines = p.apply(TextIO.read().from("hdfs://path"));
5
6 PCollection<KV<String, Integer>> words = lines.apply(ParDo.of(new DoFn<
7     String, KV<String, Integer>>() {
8     @ProcessElement
9     public void processElement(ProcessContext c) {
10         String[] words = c.element().split(" ");
11         for (String word : words) {
12             c.output(KV.of(word, 1));
13         }
14     }
15 }));
16 PCollection<KV<String, Iterable<Integer>>> groups = words.apply(
17     GroupByKey.<String, Integer>create());
18 PCollection<KV<String, Integer>> counts = groups.apply(
19     Combine.<String, Integer>groupedValues(
20     new Sum.SumIntegerFn()));
21
22 counts.apply(TextIO.write().to("wordcounts")).run().waitUntilFinish();

```

Figura 3.10 – Exemplo de aplicação de contagem de palavras no Apache Beam.

mas uma vez que seu modelo de programação é independente do MapReduce, este pode ser executado em diferentes sistemas de processamento de dados.

No Apache Beam foi apresentado o conceito de executores (*runners*), que separa os pipelines da plataforma que vai executá-los. Estes fazem com que um pipeline possa ser desenvolvido totalmente independente de onde vai ser executado e ao mesmo tempo permite que este seja executado em vários sistemas de processamento de dados diferentes. Atualmente, o Apache Beam possui runners para diferentes sistemas como o Google Clud Dataflow, Apache Spark, Apache Flink e o Apache Hadoop, além de outros.

3.5 Apache Spark

O *Apache Spark* é um framework de propósito geral para processamento paralelo de dados em cluster de computadores (SPARK, 2019). Spark foi criado como uma alternativa ao modelo MapReduce ao oferecer processamento em memória (ZAHARIA et al., 2010). Este é mais adequado para aplicações iterativas, como algoritmos de aprendizagem de máquina, e análises interativas, como consultas exploratórias em conjuntos de dados, que são limitações no modelo MapReduce uma vez que esse escreve os resultados de cada etapa em disco. Spark permite que aplicações de Big Data sejam escritas nas linguagens de programação Scala, Java, Python e R. Além disso, Spark oferece bibliotecas próprias para trabalhar com dados estruturados utilizando uma API com estilo SQL

(*SparkSQL*), aprendizado de máquina (*MLlib*), processamento de fluxo contínuo de dados (*Spark Streaming*) e processamento de grafos (*GraphX*) (SPARK, 2019).

A principal abstração do modelo de Spark é o *Resilient Distributed Dataset* (RDD), uma coleção de dados particionados nos computadores de um cluster que podem ser processados em paralelo (ZAHARIA et al., 2012). Um RDD é tolerante a falhas, o que significa que suas partições podem ser reconstruídas caso ocorra alguma falha no processo de execução. Para isso, RDDs utilizam um sistema de linha do tempo que rastreia a sequência de operações que devem ser executadas para reconstruir uma partição quando dados são perdidos.

Uma aplicação Spark é um programa (comumente chamado de *Driver*) que executa uma função principal contendo uma sequência de operações em RDDs. O Driver então se conecta com o cluster para executar essas operações em paralelo (SPARK, 2019). No Driver deve ser criado um contexto Spark (*SparkContext*), que é responsável por gerenciar informações internas da aplicação e se conectar com o cluster (GANELIN et al., 2016). Inicialmente um RDD pode ser criado de duas formas: *i*) ao referenciar um conjunto de dados em algum sistema de armazenamento, como o sistema de arquivos distribuídos HDFS, por exemplo; ou *ii*) ao paralelizar no cluster alguma coleção de dados do Driver, como listas e *arrays* (SPARK, 2019). A partir desses RDDs iniciais é possível aplicar uma série de operações Spark para compor uma aplicação de processamento de dados.

3.5.1 Operações

Operações em Spark podem ser classificadas de duas formas de acordo com o tipo dos seus resultados: *transformações* e *ações*. Transformações são o grupo de operações que criam novos RDDs a partir de outros já existentes (SPARK, 2019). Em Spark, transformações são avaliadas de maneira tardia (*lazy evaluation*) (ZAHARIA et al., 2010), de modo que elas só são computadas quando necessário. Dessa forma, quando uma transformação é chamada ela não é computada de fato, mas sua operação é registrada para ser realizada posteriormente. Muitas das operações de Spark recebem funções como parâmetro de entrada e aplicam essas funções sobre os elementos do RDD para gerar os elementos do novo RDD. Além disso, existe um grupo de transformações que só são habilitadas em RDDs de tuplas chave/valor (operações que geralmente terminam com o sufixo *byKey*). Essas transformações operam sobre os valores agrupados pela chave. Algumas das principais transformações de Spark são descritas a seguir:

map mapeia cada item do RDD para um novo valor com base em uma função de transformação passada como parâmetro e retorna um novo RDD com os valores gerados.

flatMap opera de forma semelhante ao *map*, mas a função passada como parâmetro pode mapear um item para zero ou mais novos valores e o RDD resultante possui todos os valores gerados. Essa transformação é semelhante à etapa de *Map* no modelo MapReduce.

filter filtra os itens do RDD com base em uma função de predicado passada como parâmetro. O RDD resultante possui todos os itens que foram avaliados como verdadeiro pela função.

groupByKey agrupa os itens de um RDD de tuplas chave/valor com base na chave, de forma que o RDD resultante possui tuplas com chave e uma coleção com os valores que tinham a chave em comum. Essa transformação possui o mesmo comportamento da etapa de *shuffle* do modelo MapReduce.

reduceByKey agrega os itens de um RDD de tuplas chave/valor com base na chave através de uma função de agregação passada como parâmetro. Essa função recebe dois valores como entrada e retorna um novo valor do mesmo tipo como saída. O RDD resultante possui tuplas chave/valor em que o valor é a agregação de todos os valores que eram associados com essa chave.

aggregateByKey também agrega os itens de um RDD de tuplas chave/valor com base na chave, mas possui um comportamento mais genérico que o *reduceByKey* ao permitir que os valores sejam agregados para um tipo diferente. Para isso, essa transformação recebe como parâmetro um valor inicial do novo tipo, uma função que agrega um valor do novo tipo com um valor do tipo antigo e uma função que agrega dois valores do novo tipo.

distinct retorna um novo RDD que contem apenas elementos distintos do RDD inicial.

union produz a união do conteúdo de dois RDDs de modo que o RDD resultante possui todo o conteúdo dos dois RDDs. Essa operação é semelhante à operação de união de conjuntos matemáticos, entretanto ela não garante que o RDD resultante não vai ter elementos repetidos, fazendo com que seja necessário chamar a operação *distinct* para garantir essa propriedade. Spark também possui as transformações *subtract* e *intersect* que são baseadas nas operações de diferença e interseção de conjuntos matemáticos, sendo que esta última garante que o RDD resultante não possui elementos duplicados.

join realiza a junção entre dois RDDs de tuplas chave/valor em que as chaves são do mesmo tipo. O RDD resultante possui tuplas chave/valor em que o valor é uma tupla contendo valores dos dois RDDs iniciais que possuíam a chave em comum.

sortByKey ordena os itens de um RDD de tuplas chave/valor de acordo com a chave. O RDD resultante pode ter seus itens ordenados de forma ascendente ou descendente.

repartition reorganiza os itens do RDD em um novo número de partições que pode ser maior ou menor que o número inicial. Essa reorganização é feita de maneira aleatória e todos os itens são redistribuídos através da rede. O RDD resultante possui o mesmo conteúdo que o RDD inicial, mas com uma distribuição de partições diferente.

coalesce opera de forma semelhante ao *repartition*, mas quando o novo número de partições é menor que o inicial, ela une partições que estão em um mesmo computador no cluster ao invés de redistribuir os dados na rede.

Transformações podem ser estreitas (*narrow*) ou amplas (*wide*) dependendo do grau de distribuição dos conjuntos de dados (ZAHARIA et al., 2012). Transformações estreitas são aquelas em que cada partição do RDD de origem só é utilizada por no máximo uma partição do RDD novo gerado pela operação. Operações como *map*, *flatMap* e *filter* são exemplos de transformações estreitas. Já transformações amplas são aquelas em que cada partição do RDD de origem é utilizada para criar múltiplas partições do novo RDD. Geralmente, esse tipo de transformação opera sobre conjuntos de dados de tuplas chave/valor que precisam ter seus dados agrupados por chave, como as operações *reduceByKey* e *join*, por exemplo. Esse tipo de operação causa uma reorganização dos dados no cluster de modo a agrupar valores que possuem uma mesma chave em uma mesma partição. Assim como no modelo MapReduce, essa reorganização nos dados é chamada de *shuffle*. Esse mecanismo envolve copiar e enviar dados entre os nós do cluster, fazendo com que seja um processo complexo e custoso (SPARK, 2019). A Figura 3.11 mostra uma representação de uma transformação estreita (*Narrow Transformation*) e uma transformação ampla (*Wide Transformation*). Nela as setas pretas indicam a dependência entre partições do RDD de origem (na parte de cima da figura) e as partições do RDD resultante da transformação (na parte de baixo da figura).

O segundo grupo de operações Spark são as ações. Diferentemente das transformações, ações retornam valores que não são RDDs para o Driver ou escrevem o conteúdo do RDD em algum sistema de armazenamento. Uma vez que as ações resultam em resultados concretos, essas são as operações que disparam a execução das transformações uma vez que estas só são avaliadas quando necessário (*lazy evaluation*). Com esse sistema, o Spark consegue otimizar a execução das aplicações ao executar transformações estreitas em um mesmo processo (*pipeline*) e criar diferentes estágios para operações amplas que desencadeiam o processo de *shuffle* (ZAHARIA et al., 2012). Algumas das principais ações são descritas a seguir:

reduce agrega os elementos de um RDD em um único valor através de uma função de agregação passada como parâmetro. Esta função recebe dois valores e retorna um valor do mesmo tipo. Devido a sua execução em paralelo, a função passada como

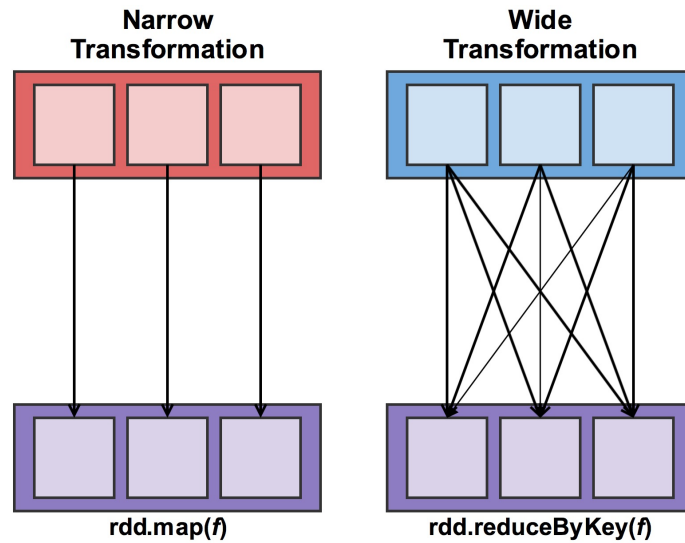


Figura 3.11 – Representação de transformação estreita e ampla.

parâmetro deve respeitar as propriedades de comutatividade e associatividade para que o resultado seja computado de forma correta.

collect retorna todo o conteúdo do RDD como uma *array* para o programa Driver.

count retorna o número de elementos no RDD.

saveAsTextFile escreve o conteúdo de um RDD para arquivos em algum sistema de arquivos. O conteúdo do RDD é salvo em várias partes de acordo com o seu número de partições. O conteúdo pode ser salvo em sistemas de arquivos distribuídos como o HDFS (SHVACHKO et al., 2010).

Uma vez que RDDs são computados apenas quando uma ação é chamada, se um RDD for utilizado por mais de uma ação, este vai ser computado novamente para cada chamada. Para evitar esse tipo de repetição, Spark permite que um RDD seja persistido de tal modo que ele é computado apenas uma única vez, com a chamada da primeira ação, e seu conteúdo é salvo para ser utilizado de forma direta nas ações subsequentes. Spark permite diferentes níveis de persistência, como persistência em memória, em disco ou em ambos, além de permitir que os dados sejam serializados ou não.

A Figura 3.12 apresenta o exemplo de uma aplicação de contagem de palavras no Apache Spark. A aplicação é iniciada com a leitura de um conjunto de dados na linha 1. Em seguida, é feita a aplicação da transformação *flatMap* que separa cada linha de texto em palavras (linha 2). O RDD com palavras é transformado em um RDD do tipo chave/valor com a aplicação da transformação *map* que gera pares chave/valor em que a chave é a palavra e o valor o número inteiro 1 (linha 3). A contagem de palavras é feita com a aplicação da transformação *reduceByKey* que agrupa os valores por chave e depois

aplica uma função associativa que soma todos os valores e que resulta na frequência de cada palavra (linha 4). A aplicação é finalizada com a chamada da ação *saveToFile* que executa as transformações e salva o seu conteúdo (linha 5).

```
1  val texts = sc.textFile("hdfs://...")
2  val words = texts.flatMap(line => line.split(" "))
3  val pairs = words.map(word => (word, 1))
4  val counts = pairs.reduceByKey((a, b) => a + b)
5  counts.saveAsTextFile("hdfs://...")
```

Figura 3.12 – Exemplo de aplicação Spark de contagem de palavras.

3.5.2 Variáveis Compartilhadas

O modelo de computação paralela de Spark requer que cada nó no cluster tenha cópias dos processos que serão executados. Logo, função e valores passados como parâmetros para as operações Spark devem ser copiados e enviados entre os nós do cluster. Em situações em que o cluster possui um grande número de computadores ou o tamanho das variáveis é grande, esse processo pode ser custoso devido ao grande número de cópias e transferências que precisam ser feitas.

Além disso, uma vez que cada nó do cluster possui cópias locais dessas variáveis, qualquer modificação feita em alguma variável não é propagada de volta para o Driver (SPARK, 2019). Esse comportamento livre de efeitos colaterais é o desejado na maioria dos casos, entretanto certos tipos de aplicações podem requerer que modificações em variáveis sejam propagadas para o Driver. Exemplos de aplicações desse tipo são as aplicações iterativas que utilizam uma variável para controlar o número de iterações. Visando resolver esses dois tipos de problemas, o Spark fornece dois tipos de variáveis compartilhadas, as variáveis de transmissão (*Broadcast Variables*) e os acumuladores (*Accumulator*) (ZAHARIA et al., 2010).

Quando uma variável é copiada e enviada para um nó no cluster, esta não é copiada uma única vez para cada nó, mas sim um número de vezes igual ao número de tarefas (*tasks*) que serão executadas no nó, em que o número de tarefas depende do número de partições do RDD. Isso significa que cada nó do cluster possui várias cópias de uma mesma variável, algo que é custoso em casos de muitas tarefas e variáveis com tamanho pesado. As variáveis de transmissão são o meio fornecido por Spark para evitar um número alto de cópias de uma variável e reduzir a sobrecarga que estas causam. Com uma variável de transmissão, ao invés de Spark criar uma cópia para cada tarefa que vai ser executada no nó, Spark cria uma única cópia no nó e permite que diferentes tarefas tenham acesso a essa cópia (SPARK, 2019). Uma vez que variáveis de transmissão são acessadas por diferentes processos, essas não podem ser modificadas.

Acumuladores são variáveis que podem ser modificadas, através de operações que adicionam novos valores a elas, e ter seus valores propagados dos nós do cluster para o Driver. Uma vez que acumuladores são modificados de forma paralela, eles suportam apenas operações comutativas e associativas (SPARK, 2019). Geralmente, acumuladores são utilizados para implementar contadores que são atualizados nos nós do cluster e utilizados no Driver.

3.5.3 Cluster Spark

Quando um programa Driver se conecta com o cluster para executar uma aplicação, este interage com o gerenciador do cluster (*Cluster Manager*), trabalhadores (*Workers*) e processos executores (*Executors*). A interação entre o Driver e o cluster é representada na Figura 3.13. O gerenciador do cluster é responsável por coordenar os computadores do cluster (nós) e alocar os seus recursos, como CPU e memória. Spark possui um gerenciador de cluster próprio (*Standalone Cluster*), mas também permite que outros gerenciadores sejam utilizados, como o *Hadoop Yarn* (HADOOP, 2019) e o *Apache Mesos* (MESOS, 2019). Os outros componentes são agnósticos com relação ao gerenciador do cluster, significando que o gerenciador utilizado não faz diferença para eles.

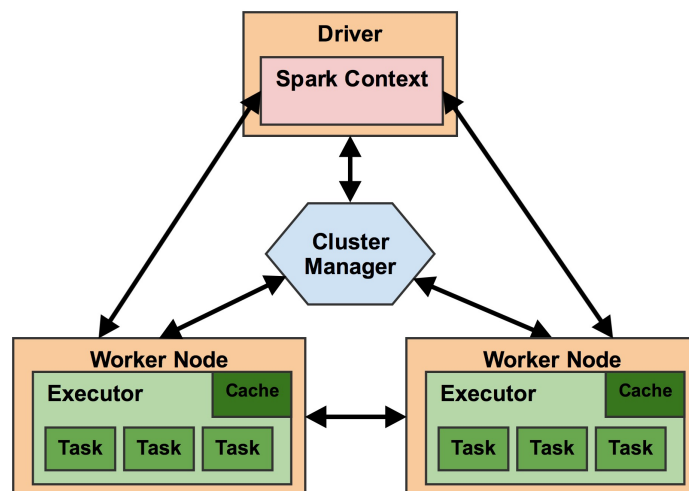


Figura 3.13 – Visão geral do cluster.

Os trabalhadores são os nós do cluster que executam processos executores e tarefas (GANELIN et al., 2016). Cada nó possui uma quantidade fixa de recursos que podem ser utilizados pelos executores. Cada processo executado possui uma instância da *Java Virtual Machine* (JVM) e encapsula uma quantidade limitada dos recursos do nó para ser utilizado na execução de tarefas. O número de executores e a quantidade de recursos alocados para eles são parâmetros escolhidos quando uma aplicação é submetida para execução no cluster. A forma em que o cluster é configurado para executar uma aplicação tem um impacto direto em seu desempenho (GANELIN et al., 2016).

3.5.4 Execução

Quando uma aplicação é submetida para execução, o programa Driver se comunica com o gerenciador do cluster para requisitar recursos para ela ser executada (GANELIN et al., 2016). Para cada ação chamada na aplicação, é criado um *job* que possui um plano de execução com base nas operações chamadas nos RDDs. Esse plano de execução é representado como uma DAG de transformações que precisam ser computadas para que a ação seja executada. Essa DAG representa a “linha do tempo” que é utilizada por Spark para rastrear as operações que precisam ser re-executadas em casos de falha.

Como forma de otimização, a DAG é separada em estágios. O número de estágios depende do número de transformações amplas que são feitas na aplicação. Transformações estreitas que são executadas em sequência são agrupadas em um mesmo estágio e transformações amplas que requerem um processo de *shuffle* geram estágios diferentes. Cada estágio é então separado em tarefas para serem executadas nos processos executores. O número de tarefas depende do número de partições do RDD e cada tarefa é processada no mesmo nó em que sua partição está alocada (GANELIN et al., 2016). Todas as tarefas executam uma mesma operação em partições diferentes no cluster e seus resultados são unificados posteriormente em um único RDD (GANELIN et al., 2016).

3.6 Discussões

Os modelos e sistemas para processamento paralelo de Big Data apresentados possuem uma série de diferenças e semelhanças. Uma vez que nosso foco é principalmente no modelo de programação, vamos comparar os sistemas apresentados em três aspectos: fluxo de dados, abstrações e interface de programação.

O MapReduce que é implementado pelo Apache Hadoop apresenta o modelo mais simples de todos ao sintetizar o programa em duas funções fixas de Map e Reduce, que consomem e produzem pares chave/valor como entrada e saída. Como dito anteriormente, esse modelo se apresenta muito limitado para representar cargas de trabalho mais complexas que podem exigir uma quantidade maior de operações que não podem ser expressas em um único programa MapReduce. Dessa forma, todos os outros sistemas apresentados neste trabalho possuem modelos mais flexíveis que não limitam a quantidade de operações em uma aplicação.

No Dryad, programas são representados como grafos acíclicos orientados (DAG) em que cada vértice executa uma função definida pelo desenvolvedor. Cada vértice pode receber um número arbitrário de entradas e produzir um número arbitrário de saídas. Essa característica faz com que o Dryad tenha o modelo mais flexível de todos. Apesar disso, esse modelo aumenta a complexidade do desenvolvimento porque exige que todo o fluxo de dados da aplicação seja definido de maneira explícita pelo desenvolvedor. Para

simplificar o desenvolvimento nesta plataforma, o DryadLINQ adiciona uma camada de abstração sob o Dryad. Nele, um conjunto de dados distribuídos é abstraído como uma única coleção (*DryadTable*) que pode ser manipulada através de uma grande quantidade de operações de alto nível (LINQ). No DryadLINQ, um programa é definido através de uma sequência de operações sobre coleções que tem origem em alguma fonte de dados, como algum sistema de arquivos distribuídos ou banco de dados SQL. Este programa é, então, compilado de maneira eficiente para uma DAG do Dryad, fazendo com que o desenvolvedor não tenha que definir o fluxo de dados manualmente.

O sistema Nephele opera de forma semelhante ao Dryad em termos de modelo de programação, de forma que seus programas também são representados como DAGs e seus vértices são funções definidas pelo usuário. Os PACTs adicionam uma camada de abstração sob o Nephele ao expandir o fluxo de dados e operações do modelo MapReduce. Um PACT é definido por um contrato de entrada, que define o tipo de operação que será realizada ao indicar como os dados de entrada são representados, uma função definida pelo desenvolvedor que processa esses dados, e um contrato opcional de saída, que define propriedades sobre os resultados da função que podem ser utilizadas no processo de otimização. Um programa é definido como uma sequência de PACTs que operam sobre conjuntos de dados do tipo chave/valor. Um programa PACTs é, então, compilado para uma DAG otimizada para ser executada no sistema Nephele.

O FlumeJava foi proposto como uma interface de alto nível que simplificava o desenvolvimento de *pipelines* (sequências de operações) no modelo MapReduce. Este deu origem ao Apache Beam, que expandiu os conceitos do FlumeJava para uma interface unificada que permite o desenvolvimento de pipelines em diferentes sistemas para processamento de Big Data. No Apache Beam, um conjunto de dados distribuídos é abstraído como uma única coleção (*PCollection*). Essa coleção é processada através de operações que transformam uma coleção em outra (*PTransform*). Um programa (*Pipeline*) é definido a partir de uma coleção de dados iniciais, gerados a partir de alguma fonte de dados, uma sequência de transformações sobre essas coleções e uma operação final que salva os resultados do processamento em alguma fonte externa. O plano de execução de um programa no Apache Beam forma uma DAG que é definida a partir das dependência entre transformações que precisam ser executadas para computar uma coleção. Essa DAG é utilizada pelo Apache Beam para definir um plano de execução em diferentes plataformas (*Runners*).

O Apache Spark tem como principal abstração o RDD. Este representa um conjunto de dados particionados que podem ser processados em paralelo. RDDs são processados através de dois tipos de operações: transformações e ações. Transformações são operações que geram novos RDDs a partir de outros. Já ações são operações que geram algum resultado diferente de um RDD, como resultar em algum valor agregado ou salvar

o conteúdo do RDD em alguma fonte externa. No Spark, transformações são avaliadas de forma tardia (*lazy*), de forma que só são computadas após a chamada de alguma ação. Dessa forma, uma aplicação Spark é definida a partir de um conjunto de RDDs iniciais, que são criados a partir de alguma fonte de dados, uma sequência de transformações em RDDs e uma ação que finaliza o processamento. Essa sequência de operações é representada como uma DAG que forma um plano de execução baseado na dependência entre transformações.

Com exceção do modelo MapReduce, todos os modelos apresentados tem seus programas representados através de DAGs. De maneira geral, os vértices representam operações que fazem algum tipo de processamento no conjunto de dados. Os resultados do processamento de um vértice são passados para os seguintes até chegar ao último que finaliza o processamento. A forma em que os dados são transportados entre os vértices (operações) pode variar de acordo com o sistema. No Dryad e Nephelê/PACTs, os dados são transportados através de canais de comunicação, que são formas de armazenamento intermediário para o qual um vértice lê os dados de entrada e escreve os dados de saída. Já o DryadLINQ, Apache Beam e Apache Spark, optaram por criar uma abstração para conjuntos de dados distribuídos e representar um programa através de operações que vão fazendo transformações nesses conjuntos de dados. Dessa forma, é possível dizer que o *DryadTable* (DryadLINQ), *PCollection* (Apache Beam) e *RDD* (Apache Spark) possuem conceitos análogos.

Com relação à interface de programação, os sistemas Dryad e o Nephelê oferecem um modelo flexível ao permitir que operações possam ser definidas sem nenhuma interface fixa. Dessa forma, operações definidas pelo desenvolvedor podem receber um número arbitrário de entradas e produzir um número arbitrário de saídas. Além disso, a comunicação entre operações deve ser definida de maneira explícita, o que pode dificultar a programação. Os outros sistemas optaram por fornecer interfaces de alto nível em que as operações possuem um tipo de entrada, processamento e saída bem definidos. Esse tipo de interface simplifica o desenvolvimento porque permite que o desenvolvedor se foque em operações mais específicas e não exige que ele administre detalhes de baixo nível, como adequar o tipo de saída de uma operação com a entrada de outra.

Ao analisar as operações disponíveis em cada sistema, é possível ver que existem várias semelhanças em relação aos tipos de operações. A Tabela 3.1 apresenta uma comparação entre os tipos de operações disponíveis nos sistemas. Na tabela, as operações são classificadas como: *Mapeamento*, que representa operações que mapeiam um valor para outros a partir de alguma função de transformação; *Filtragem*, que representa operações que removem valores de uma coleção com base em algum predicado; *Agrupamento*, que representa operações que agrupam valores com base em uma chave; *Agregação*, que representa operações que sintetizam um grupo de valores em um único valor; *Conjuntos*, que

representa operações que são inspiradas nas operações de conjuntos matemáticos; *Junção*, que representa operações que fazem uma junção relacional entre dois conjuntos de dados; e *Ordenação*, que representa operações que ordenam os valores do conjunto de dados.

Tabela 3.1 – Comparação das interface de programação de sistemas de processamento de Big Data.

	MapReduce	DryadLINQ	PACTs	Apache Beam	Apache Spark
Mapeamento	Map	Select, Select-Many	Map	ParDo	map, flatMap
Filtragem	~	Where	~	~	filter
Agrupamento	~	GroupBy	~	GroupByKey	groupByKey
Agregação	Reduce	Aggregate	Reduce, CoGroup	Combine	reduce, reduceByKey, aggregateByKey
Conjuntos	~	Union, Intersect, Except, Distinct	Cross	Flatten	union, intersection, subtract, distinct
Junção	~	Join	Match	CoGroupByKey	join
Ordenação	~	OrderBy	~	~	sortByKey

Na Tabela 3.1 é possível ver que todos os sistemas possuem operações equivalentes às operações Map (Mapeamento) e Reduce (Agregação) do modelo MapReduce, de modo que se torna fácil escrever um programa no estilo MapReduce em qualquer um dos outros modelos. Os sistemas que fornecem a maior quantidade de operações são o DryadLINQ e o Apache Spark, possuindo operações em todos os tipos seguindo a classificação utilizada na tabela. Já o PACTs e Apache Beam também possuem uma quantidade ampla de operações que podem ser compostas para a criação de operações mais complexas.

4 Uma Taxonomia de Problemas de Desempenho para Apache Spark

Este capítulo apresenta o resultado de um estudo sobre problemas de desempenho em aplicações Spark. A partir desse estudo, foi desenvolvida uma taxonomia¹ que agrupa e categoriza problemas que podem afetar negativamente o desempenho de execução de uma aplicação Spark. As fontes desses problemas vão desde decisões de projeto no desenvolvimento das aplicações à critérios de ajustes de configuração no cluster que vai executar a aplicação. O objetivo dessa taxonomia é modelar problemas de desempenho de execução em aplicações Spark e fornecer um guia para desenvolvedores e administradores de cluster sobre aspectos que devem ser levados em consideração durante o desenvolvimento ou execução de uma aplicação Spark com o objetivo de se ter um melhor desempenho.

Este capítulo está organizado da seguinte forma: a Seção 4.1 apresenta a taxonomia de problemas de desempenho para Spark; a Seção 4.2 apresenta os experimentos realizados para ilustrar os problemas de desempenho descritos na taxonomia; e a Seção 4.3 apresenta uma discussão dos resultados dos experimentos.

4.1 Taxonomia

A taxonomia é dividida em seis categorias de problemas de desempenho como mostrados na Figura 4.1: *Gerenciamento de Recursos do Cluster*; *Particionamento dos Dados*; *Transmissão de Dados*; *Persistência de Dados*; *Redistribuição de Dados*; e *Projeto da Aplicação*. Essas categorias e os problemas descritos em cada uma são apresentados a seguir.

I1 - Gerenciamento de Recursos do Cluster: este grupo descreve problemas que são relacionados ao alocamento dos recursos do cluster para executar uma aplicação Spark. Os recursos considerados são o número de computadores (*workers*) no cluster e a quantidade de núcleos de CPU e memória RAM disponíveis em cada computador. Ter um bom desempenho na execução de uma aplicação esbarra no desafio de encontrar a configuração que melhor equilibra os recursos disponíveis no cluster com a carga de trabalho exigida pela aplicação que será executada (GANELIN et al., 2016).

¹ A Taxonomia de Problemas de Desempenho e os resultados de seus experimentos deram origem a um artigo intitulado “A Taxonomy of Performance Issues for Apache Spark”. Este artigo foi escrito em parceria com os professores Dr. Martin Alejandro Musicante, Dr^a. Genoveva Vargas-Solar e Dr^a. Anamaria Martins Moreira. O artigo foi submetido para o *Journal Cluster Computing* e no momento se encontra em fase de avaliação.

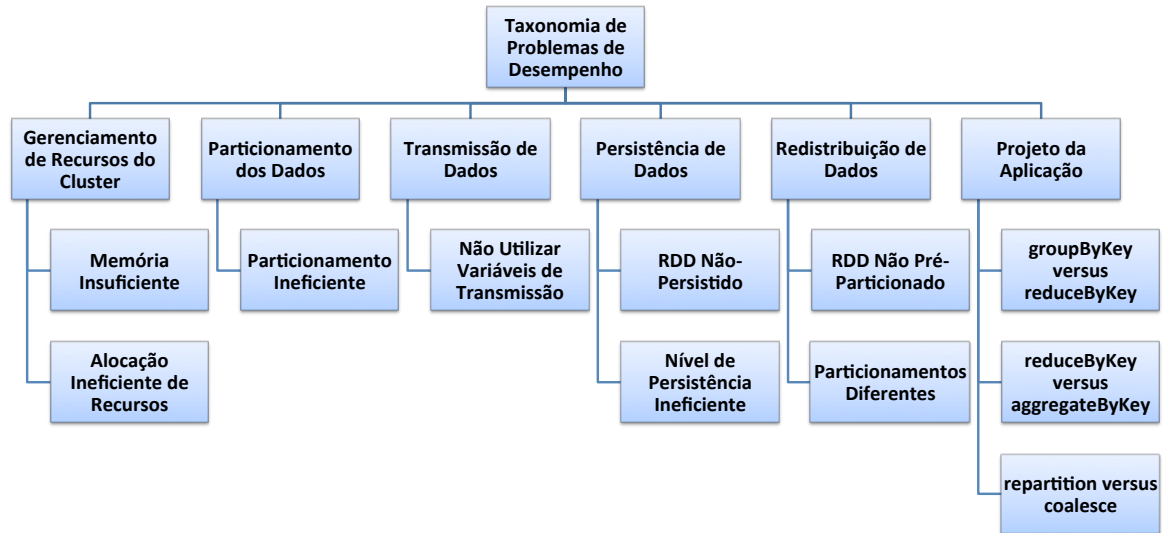


Figura 4.1 – Taxonomia de Problemas de Desempenho.

I1.1: Memória Insuficiente: quando uma aplicação é submetida para execução no cluster, o desenvolvedor ou administrador do cluster deve destinar uma quantidade de memória RAM suficiente para a execução. Alocar uma quantidade insuficiente de memória pode causar perda de desempenho por diferentes motivos. Quando um RDD é persistido em memória para ser reutilizado em diferentes ações, por exemplo, se a memória RAM não for suficiente para armazenar todo o conteúdo do RDD, Spark faz com que apenas parte desse conteúdo seja salvo e computa a parte não salva novamente sempre que necessário (SPARK, 2019), algo que pode lidar com uma sobrecarga na execução e perda de desempenho. Além disso, alocar pouca memória para os executores pode causar erros por falta de memória devido à grande quantidade de dados que estão sendo processados ou perda de desempenho quando dados são transferidos entre disco e memória (GANELIN et al., 2016).

Por exemplo, vamos considerar um cluster com apenas dois nós (*workers*) em que cada computador possui 16 núcleos de CPU e 8 GB de memória RAM disponíveis para execução. Se configurarmos o cluster para executar mais de oito processos executores em cada nó, cada executor vai ter menos de 1 GB de memória RAM disponível. Essa quantidade é menor que a disponível na configuração padrão de Spark (1 GB) (SPARK, 2019), o que pode ter um impacto negativo no desempenho da aplicação.

I1.2: Alocação Ineficiente de Recursos: a estratégia utilizada para alocar os recursos do cluster para executar uma aplicação tem impacto direto no seu desempenho. Essa estratégia envolve determinar o número de processos executores e quantidade de memória RAM e núcleos de CPU alocados para cada executor. Mesmo nos casos em que todos os recursos do cluster estão disponíveis para uma aplicação, o desempenho depende das características da aplicação e conjunto de dados que será processado e se a quantidade de executores foi configurada de maneira equilibrada.

Vamos considerar novamente o exemplo de um cluster com dois nós com 16 núcleos de CPU e 8 GB de memória RAM cada. É possível configurar esse cluster para executar uma aplicação de diferentes formas. Por exemplo, é possível configurar o cluster para ter um processo executor com 16 núcleos de CPU e 8 GB de memória RAM em cada nó, ou ter dois processos executores com 8 núcleos de CPU e 4 GB de memória em cada nó, além de outras possíveis configurações. Em ambos os casos, todos os recursos do cluster estariam disponíveis para a aplicação, entretanto, o desempenho pode ser diferente para cada configuração. Essa influência do número de executores no desempenho pode ser vista nos casos em que o conjunto de dados está sendo lido a partir de um sistema de arquivos distribuídos como o HDFS (GANELIN et al., 2016), por exemplo. O número de operações simultâneas suportadas pelo HDFS depende do número de partições do conjunto de dados em cada nó do cluster. Dependendo do tamanho do conjunto de dados e de como este está distribuído através do cluster, uma configuração com menos executores pode ter um pior desempenho porque fará com que uma quantidade maior de núcleos de CPU em um mesmo executor tenham que ler os dados de uma mesma partição no nó. Em contrapartida, o caso com um maior número de executores e uma menor quantidade de núcleos de CPU por executor pode ter um melhor desempenho porque vai ter uma quantidade menor de núcleos lendo uma mesma partição, o que por sua vez potencializa o uso dos núcleos porque se tem uma quantidade maior de núcleos lendo diferentes partições (GANELIN et al., 2016).

I2 - Particionamento dos Dados: o número de partições de um RDD e o número de núcleos de CPU alocados para a aplicação determinam a quantidade de dados que podem ser processados em paralelo. A estratégia de particionamento do RDD pode vir junta com o conjunto de dados, algo que acontece ao se ler dados do HDFS, por exemplo. Uma vez que o HDFS particiona automaticamente o conjunto de dados em vários blocos, Spark aproveita essa configuração e cria uma partição no RDD para cada bloco no HDFS (SPARK, 2019). Entretanto, essa estratégia de particionamento também pode ser configurada ou programada de forma explícita através da definição do número de partições que um RDD vai ter. Essa estratégia pode ser ajustada de acordo com o número de núcleos de CPU disponíveis para a aplicação com o objetivo de maximizar seu uso.

I2.1: Particionamento Ineficiente: a quantidade de partições de um conjunto de dados determina o tamanho de cada partição e o número de tarefas que Spark executa em paralelo. Isso ocorre porque Spark agenda e executa uma única tarefa para processar os dados de cada partição (KARAU; WARREN, 2017). A quantidade de núcleos de CPU disponíveis para a aplicação restringe o número de tarefas que podem ser executadas em paralelo. Dessa forma, quando um conjunto de dados é dividido em poucas partições, a aplicação pode não usufruir de todo potencial do cluster uma vez que Spark pode deixar recursos ociosos no caso que tiver uma quantidade menor de partições para serem

processadas do que núcleos de CPU disponíveis (KARAU; WARREN, 2017). Além disso, ter uma quantidade menor de partições faz, conseqüentemente, com que cada partição tenha uma quantidade maior de dados, o que pode pressionar a memória nas tarefas e no *garbage collector* (GANELIN et al., 2016). Em contrapartida, a situação contrária, em que um conjunto de dados está dividido em muitas partições, pode fazer com que a aplicação sofra com sobrecarga porque ter um número de partições muito maior que o número de núcleos de CPU faz com que tarefas tenham que aguardar recursos ficarem disponíveis para poderem ser processadas (KARAU; WARREN, 2017).

A escolha do número de partições em relação ao número de núcleos de CPU disponíveis não é trivial, como mostrado em (LI et al., 2017). Escolher o número de partições exatamente igual ao número de núcleos pode não ser eficiente porque dependendo do tamanho do conjunto de dados, ter uma partição com muitos dados pode sobrecarregar a memória durante a sua execução da tarefa. Nos experimentos realizados em (LI et al., 2017), os melhores desempenhos foram obtidos ao se escolher uma quantidade de 1.5 tarefas (ou partições) por núcleo de CPU. Na documentação de Spark (SPARK, 2019), é recomendado se ter de duas à quatro tarefas por núcleo. Com base nesses valores, no exemplo do cluster que contem 16 núcleos de CPU apresentado anteriormente, o número de partições ideal deve ser entre 24 (1.5 vezes) e 64 (4 vezes) partições.

I3 - Transmissão de Dados: a troca de dados entre o programa Driver e os executores pode causar uma sobrecarga na execução porque Spark precisa enviar cópias desses dados para cada tarefa que será executada. Dessa forma, reduzir a quantidade de dados que será transmitido do Driver para os executores pode reduzir o impacto negativo que essa transmissão tem no desempenho da aplicação.

I3.1: Não Utilizar Variáveis de Transmissão (Unbroadcasted Variable): quando uma clausura (*closure*, uma função com referências externas) é passada como parâmetro para uma operação, Spark envia cópias de todos os valores locais referenciados por essa função para todas as tarefas criadas para essa operação. Se o número de tarefas for muito alto, Spark cria e envia cópias desses valores nessa mesma quantidade. Isso pode causar uma sobrecarga na execução, principalmente nos casos em que o tamanho dos valores for grande, e prejudicar o desempenho da aplicação.

Para reduzir essa sobrecarga, Spark fornece variáveis de transmissão (*broadcast variables*). Ao utilizar uma variável de transmissão, Spark envia uma cópia dessa variável para cada executor ao invés de enviar uma cópia para cada tarefa. Dessa forma, as tarefas podem ler a versão local do executor ao invés de ter uma cópia cada uma, o que pode reduzir consideravelmente a quantidade de dados transmitidos. Logo, não utilizar variáveis de transmissão nos casos em que existe um grande número de tarefas ou as variáveis a serem transmitidas possuem um tamanho pesado pode impactar de forma

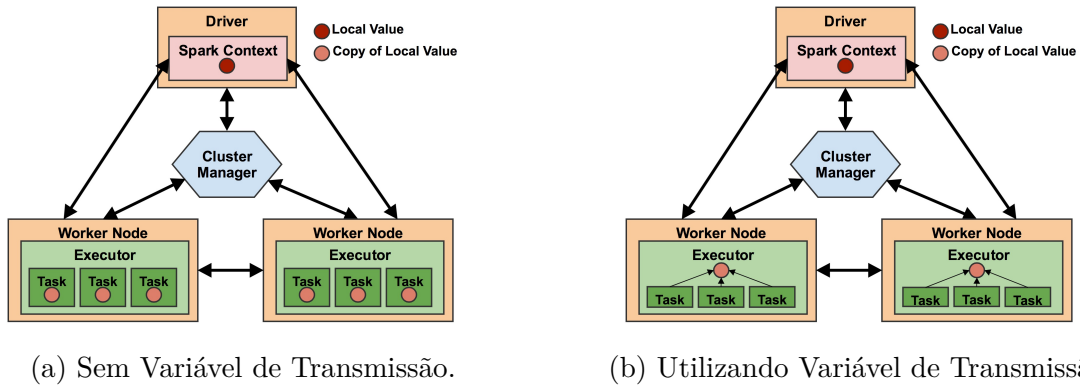


Figura 4.2 – Ilustração do problema de não utilizar variáveis de transmissão.

negativa o desempenho da aplicação.

Por exemplo, se um RDD possuir 1000 partições divididas e um cluster com 10 computadores e um executor em cada um, um valor local sem variável de transmissão seria copiado 1000 vezes, uma média de 100 vezes para cada executor. Mas utilizando uma variável de transmissão, esse valor seria copiado apenas 10 vezes. Essa situação pode ser vista na Figura 4.2. Podemos ver que no caso em que uma variável de transmissão não foi utilizada (a), cada tarefa tem sua própria cópia desse valor. No segundo caso com variável de transmissão (b), esse valor foi copiado apenas uma vez por executor e suas tarefas referenciam essa cópia local. Essa redução na quantidade de dados transmitidos pode ter um grande impacto no desempenho quanto maior for a dimensão da aplicação.

I4 - Persistência de Dados: Spark adota uma abordagem de avaliação tardia (*lazy evaluation*), dessa forma, um RDD só é computado quando alguma ação é chamada. Então, quando um mesmo RDD é utilizado em diferentes ações, este é computado novamente para cada ação chamada. Para evitar essa repetição, Spark permite que RDDs sejam persistidos em disco ou memória para quando computados uma primeira vez, não precisem ser computados novamente nas ações subsequentes. O ato de persistir um RDD ou não pode levar a problemas de desempenho dependendo do tamanho do conjunto de dados, complexidade da carga de trabalho e nível de persistência.

I4.1: RDD Não-Persistido: utilizar um RDD em mais de uma ação pode acarretar em operações redundantes devido à abordagem de avaliação tardia de Spark. Nos casos em que o RDD demanda muito esforço para ser computado, como várias operações que demandam etapas de reorganização dos dados (*shuffle*), esse trabalho redundante de computar o mesmo RDD várias vezes pode prejudicar o desempenho da aplicação.

I4.2: Nível de Persistência Ineficiente: por padrão, RDDs são persistidos em memória. Apesar disso, outros níveis de persistência estão disponíveis em Spark, tais como em disco ou em disco e memória juntos, além de permitir que os dados sejam serializados ou não. Dependendo do contexto da aplicação e do tipo do conjunto de dados, diferentes níveis

de persistência proporcionam variações no desempenho da aplicação. Consequentemente, escolher um nível de persistência errado para o contexto pode prejudicar o desempenho da aplicação ao invés de melhorar, o que é pretendido com a persistência. Escolher o melhor nível de persistência, ou seja, aquele que melhor se adéqua ao contexto de sua aplicação levando em conta o tamanho do conjunto de dados e os tipos de operações que estão sendo feitas, constitui um desafio.

I5 - Redistribuição de Dados (Data Shuffling): redistribuir dados do tipo chave/valor no cluster para agrupar valores com mesma chave em uma mesma partição é um processo custoso. Devido ao esforço para se fazer esse processo e o seu impacto no desempenho da aplicação, o uso de operações que exigem redistribuição dos dados deve ser minimizado ou evitado sempre que possível. Uma maneira de reduzir esses custos é fazer uso dos recursos que Spark disponibiliza para controlar o particionamento e, consequentemente, reduzir a quantidade de dados que precisam ser redistribuídos.

I5.1: RDD Não Pré-Particionado (Not Pre-Partitioned RDD): Spark permite que o particionamento de um RDD de dados do tipo chave/valor seja controlado de maneira explícita pelo desenvolvedor. Esse processo é feito com o uso de particionadores (*Partitioners*) que agrupam os valores que possuem uma chave comum em uma mesma partição, através de uma função que opera sob a chave para determinar para qual partição um valor vai ser distribuído. Controlar como dados do tipo chave/valor são agrupados pode reduzir a comunicação entre os computadores do cluster e, consequentemente, melhorar o desempenho da aplicação (KARAU; WARREN, 2017). Isso acontece porque operações que trabalham com chave/valor, como a operação *reduceByKey* por exemplo, se beneficia de um pré-particionamento do RDD uma vez que os valores que possuem uma chave em comum já se encontram em uma mesma partição. Dessa forma, o processo de redistribuição dos dados não é mais necessário, o que pode melhorar o desempenho da aplicação.

Essa situação é representada na Figura 4.3. Nela, é possível ver que o RDD não pré-particionado (representado em azul no lado esquerdo) realiza um processo de redistribuição (*shuffling*) para agrupar os valores com mesma chave e processar a operação *reduceByKey*. No segundo caso, o RDD pré-particionado (representado em vermelho no lado direito) não precisa fazer um processo de redistribuição. Isso faz com que ao pré-particionar um RDD de chave/valor, uma transformação ampla (*wide transformation*, transformação que exige redistribuição de dados) pode se tornar estreita (*narrow transformation*, que não exige redistribuição dos dados).

I5.2: Particionamentos Diferentes: de maneira geral, transformações que operam sob chave/valor se beneficiam de um RDD já particionado. Entretanto, isso pode ser ainda mais importante para transformações que operam sob dois RDDs, como operações de junção (*join*). Mesmo nos casos em que os dois RDDs já estão particionados, se eles

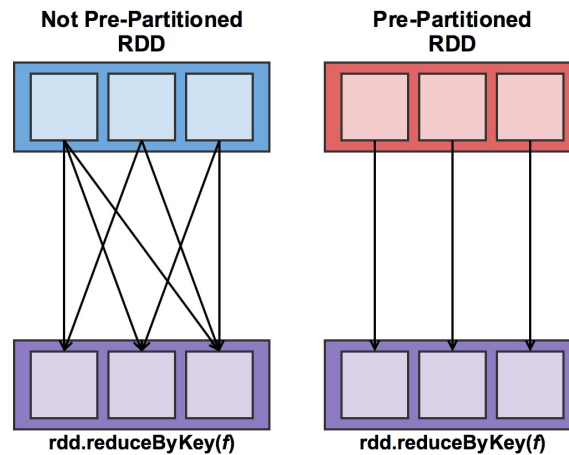
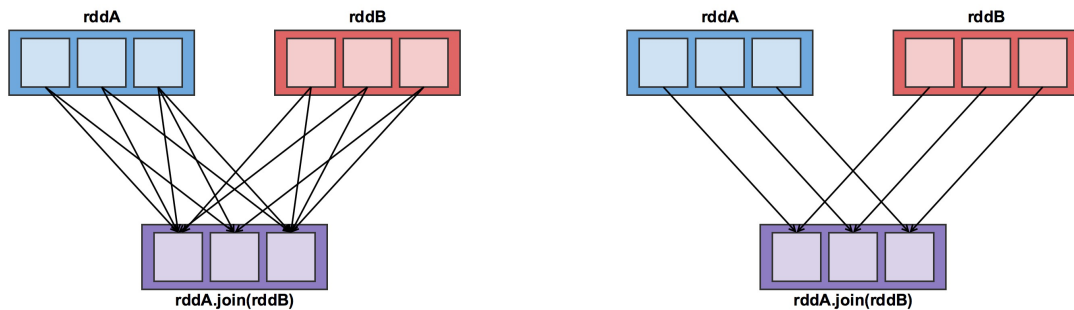


Figura 4.3 – RDD Não Pré-Particionado.



(a) Junção com RDDs não co-particionados.. (b) Junção com RDDs co-particionados..

Figura 4.4 – Particionamentos Diferentes.

foram particionados com diferentes particionadores o processo de redistribuição ainda ocorre uma vez que valores que possuem uma mesma chave em ambos os RDDs podem não estar em uma mesma partição. Dessa forma, pré-particionar ambos os RDDs utilizando o mesmo particionador antes de uma operação de junção pode reduzir mais um estágio de redistribuição e, conseqüentemente, melhorar o desempenho da aplicação.

Essa situação é representada na Figura 4.4. Podemos ver que na situação em que a junção é feita com dois RDDs que não estão co-particionados (a), o processo de redistribuição ocorre independentemente deles terem sido pré-particionados ou não. Na situação oposta (b), os valores de ambos os RDDs que possuem uma mesma chave já se encontram em uma mesma partição, fazendo com que o processo de redistribuição não seja necessário.

I6 -Projeto da Aplicação: diferentes operações de Spark podem ser utilizadas para se obter um mesmo resultado. Entretanto, mesmo realizando tarefas similares, o desempenho de cada operação pode variar consideravelmente. Dessa forma, escolher corretamente uma operação em determinadas situações pode melhorar o desempenho de uma aplicação.

I6.1: groupByKey versus reduceByKey: as operações `groupByKey` e `reduceByKey` podem

ser utilizadas para aplicar uma função a todos os valores que possuem uma mesma chave. Entretanto, a maneira em que cada operação realiza a redistribuição dos dados é diferente (GANELIN et al., 2016). Com a operação *groupByKey*, a redistribuição é feita em todo o conjunto de dados e, só após, todos os valores com uma mesma chave são processados em uma única tarefa. Já com a operação *reduceByKey*, primeiro é realizada a redução local com os valores que possuem uma mesma chave e que já se encontram em uma mesma partição. Em seguida, esses resultados intermediários são redistribuídos para serem processados com outros de mesma chave. Essa diferença no momento em que a redistribuição é feita pode reduzir consideravelmente a quantidade de dados que são trafegados no cluster, impactando no desempenho da aplicação.

As figuras 4.5 e 4.6 mostram exemplos de aplicações que utilizam as operações *groupByKey* e *reduceByKey* para realizar uma mesma tarefa. Em ambas, as operações estão sendo utilizadas para somar todos os valores que possuem uma mesma chave. Na Figura 4.5, a operação *groupByKey* é aplicada para agrupar os valores (linha 3) e, em seguida, a operação *map* é utilizada para somar todos os valores nesse grupo (linha 4). No segundo exemplo (Figura 4.6), apenas a operação *reduceByKey* é aplicada (linha 3). Ambas aplicações estão realizando a mesma tarefa e vão gerar os mesmos resultados. Apesar disso, os desempenhos serão diferentes devido às diferenças entre *groupByKey* e *reduceByKey*. É importante destacar que nem sempre as duas operações podem ser utilizadas para se fazer uma mesma tarefa.

```
1  val pairs = userVisits.map( u =>
2    (u.sourceIP, u.adRevenue))
3  val results = pairs.groupByKey()
4    .map(v => (v._1, v._2.sum))
```

Figura 4.5 – Exemplo de redução com *groupByKey*.

```
1  val pairs = userVisits.map( u =>
2    (u.sourceIP, u.adRevenue))
3  val results = pairs.reduceByKey(_ + _)
```

Figura 4.6 – Exemplo de redução com *reduceByKey*.

16.2: reduceByKey versus aggregateByKey: operações de agregação são comumente aplicadas em dados do tipo chave/valor para reduzir os valores que possuem uma mesma chave a um único valor. Em certas ocasiões, é necessário agregar os valores e modificar o seu tipo. Com Spark, isso pode ser feito de diferentes maneiras ao se utilizar as operações *groupByKey*, *reduceByKey* ou *aggregateByKey*. Entretanto, não só a forma em que cada operação administra a redistribuição dos dados é diferente, como também seu comportamento é diferente quando é necessário criar objetos de tipos diferentes para se

mudar o tipo dos valores agregados. Essa diferença na criação de objetos pode impactar no desempenho da aplicação.

Pelo motivo apresentado no problema *I6.1*, a operação *groupByKey* deve ser evitada nos casos em que se faz alguma agregação. Para utilizar a operação *reduceByKey* em agregações que envolvem mudança de tipo deve-se primeiro mapear todos os valores do RDD para o novo tipo para em seguida fazer a agregação. Essa operação faz com que novos objetos sejam criados na mesma quantidade do tamanho do RDD, o que pode acarretar em uma sobrecarga na memória e no *Garbage Collector* (GANELIN et al., 2016).

A Figura 4.7 apresentam um exemplo em que é feita uma agregação com mudança de tipo utilizando a operação *reduceByKey*. Nela, temos um RDD de dados do tipo chave/valor em que a chave é o endereço de um site (*destURL*) e o valor o IP de um usuário que visitou este site (*sourceIP*) na linha 1. Nesse exemplo, queremos obter o conjunto de usuários distintos que visitaram cada site. Para isso, mapeamos todos os usuários para um tipo de conjunto (linha 2) e em seguida unimos todos os conjuntos com *reduceByKey* (linha 3). Como o tipo de conjunto não permite repetição de objetos, obtemos um conjunto com os usuários distintos.

```
1  val userAccesses = userVisits.map(u => (u.destURL, u.sourceIP))
2  val mapUserAccess = userAccesses.map(u => (u._1, Set(u._2)))
3  val distinctSites = mapUserAccess.reduceByKey(_ ++ _)
```

Figura 4.7 – Agregação com mudança de tipo utilizando *reduceByKey*.

Em contrapartida, com a operação *aggregateByKey* é possível evitar a criação de uma grande quantidade de objetos novos e ignorar a etapa de mapeamento. Isso é possível porque a operação permite que valores de tipos diferentes sejam agregados, o que reduz a quantidade de objetos novos que precisam ser criados. A Figura 4.8 apresenta um exemplo em que se é feita a mesma aplicação que o exemplo da Figura 4.7, mas utilizando *aggregateByKey*. A operação *aggregateByKey* recebe como entrada três valores: um valor do novo tipo (linha 4), geralmente um valor identidade; uma função que agrega um valor do tipo antigo com um valor do novo tipo, gerando outro valor do novo tipo (linha 5); e uma função que agrega dois valores do novo tipo (linha 6). Com esses três parâmetros, a operação *aggregateByKey* evita que novos objetos sejam criados na mesma quantidade de objetos no RDD porque ela não precisa mapear todos os valores do RDD para o novo tipo. Com relação à redistribuição dos dados, ambas as operações trabalham de forma semelhante os redistribuir resultados intermediários ao invés de todo o conjunto de dados. Dessa forma, a operação *aggregateByKey* fornece um melhor desempenho para agregações que envolvem mudança de tipo.

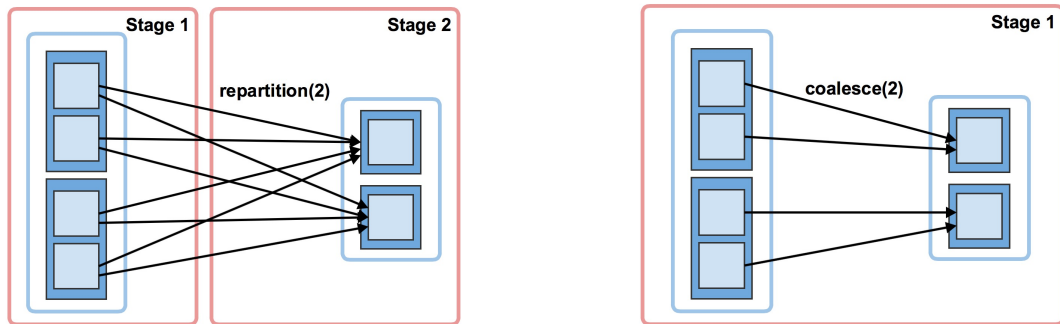
I6.3: repartition versus coalesce: em certas ocasiões, é necessário mudar o número de partições de um RDD para se evitar sobrecargas e melhorar a paralelização no processa-

```

1  val userAccesses = userVisits.map(u => (u.destURL, u.sourceIP))
2  val zeroValue = collection.mutable.Set[String]()
3  val distinctSites = userAccesses
4  .aggregateByKey(zeroValue)
5  ((set, v) => set += v,
6  (setOne, setTwo) => setOne ++= setTwo)

```

Figura 4.8 – Agregação com mudança de tipo utilizando *aggregateByKey*.



(a) Reduzindo o número de partições com *repartition*. (b) Reduzindo o número de partições com *coalesce*.

Figura 4.9 – *repartition* versus *coalesce*.

mento dos dados (GANELIN et al., 2016). Para isso, Spark disponibiliza duas operações, *repartition* e *coalesce*. A operação *repartition* redistribui os dados no cluster de maneira aleatória para um número de partições que pode ser menor ou maior que o número de partições no RDD inicial (GANELIN et al., 2016). A operação *coalesce* também tem o mesmo comportamento quando o novo número de partições é maior que o original. Entretanto, quando o número de partições é menor, a operação *coalesce* evita a redistribuição dos dados ao unir partições que se encontram em um mesmo computador no cluster. Consequentemente, a operação *coalesce* tem um melhor desempenho porque consegue reduzir o número de partições em um mesmo estágio, diferentemente de *repartition*.

A diferença entre as operações *repartition* e *coalesce* é representada na Figura 4.9. No caso em que se é utilizado *repartition* (a), as novas partições podem conter elementos de diferentes partições, que podem estar em diferentes nós do cluster, do RDD de origem. Já no caso com *coalesce* (b), esse resultado é obtido ao se unir partições que estavam em um mesmo nó, sem que a redistribuição dos dados seja necessária, o que pode acarretar em um melhor desempenho. Mas é importante destacar que ambas as operações possuem o mesmo comportamento nos casos em que o número de partições é aumentado. Além disso, em determinadas situações o comportamento da operação *repartition* pode ser desejado uma vez que este redistribui os dados de maneira equilibrada entre as partições. Uma situação em que isso pode ser necessário é quando se é feito algum tipo de filtragem no RDD e algumas partições podem ficar com um número de elementos muito diferente de outras, causando um desbalanceamento no processamento.

4.2 Experimentos

Para ilustrar a forma em que os problemas descritos na taxonomia podem aparecer em aplicações e sua influência no desempenho de execução, fizemos experimentos em um ambiente de cluster. O objetivo dos experimentos foi executar aplicações Spark que simulassem os problemas descritos na taxonomia. Em seguida, os desempenhos dessas aplicações foram comparados com o desempenho de outras versões dessas mesmas aplicações que não possuíam os problemas. Caracterizamos os experimentos realizados neste estudo descrevendo o ambiente de execução, metodologia, conjuntos de dados, aplicações e resultados.

4.2.1 Ambiente de Execução

Os experimentos foram executados em um cluster com quatro nós de *máquinas virtuais* hospedado no *Data Center* do *Instituto Metrópole Digital*. Cada nó possui 8 núcleos vCPU utilizando a CPU Intel Xeon x5650 2.67Ghz, 8 GB de memória RAM e dois discos virtuais anexados com 112 GB de espaço armazenados em discos rígidos SAS Nearline 7k - LFF (3,5"). Os nós são conectados através de uma rede LAN de 10 Gbps. Dos 8 GB de memória RAM disponíveis em cada nó, 6 GB são dedicados para Spark os outros 2 GB são utilizados pelo sistema operacional e pelo *Hadoop HDFS*, que é utilizado para armazenar os conjuntos de dados e resultados das aplicações. O *Standalone Cluster* disponibilizado por Spark foi utilizado como gerenciador do cluster nos experimentos. Dos quatro nós disponíveis, um foi configurado como o nó principal (*master*), que gerencia o cluster Spark e o sistema de arquivos distribuídos *Hadoop HDFS*. Os outros três nós são utilizados pelo nó principal (*slaves*) para executar as aplicações Spark e armazenar os dados do HDFS. A versão de Spark utilizada nesses experimentos foi a 2.2.0 (SPARK, 2019), que era a versão atual quando os experimentos foram realizados. Esse ambiente de execução permite que os problemas de desempenho descritos na taxonomia sejam investigados sob diferentes configurações.

4.2.2 Metodologia

Os experimentos foram conduzidos para mostrar a influência que os problemas descritos têm no desempenho de aplicações Spark. Os experimentos foram feitos de forma semelhante aos experimentos apresentados em (LI et al., 2017), um dos poucos trabalhos que mostraram experimentalmente a influência dos parâmetros de configuração do cluster em Spark. Uma vez que são observadas diferenças de desempenho em sucessivas execuções de uma mesma aplicação, nós executamos cada experimento por oito vezes e relatamos a média dos resultados. Para reduzir a influência de valores atípicos nos resultados (*outliers*, valores que apresentam uma grande diferença em relação aos outros), nós utilizamos a

média truncada (*trimmed mean*) (MARONNA; MARTIN; YOHAI, 2006). Essa média desconsidera parte dos maiores e menores valores que podem influenciar negativamente os resultados. Para calcular a média truncada, nós descartamos os resultados com maior e menor tempo de duração das oito execuções e computamos a média aritmética das seis execuções restantes. Nosso processo se diferencia do apresentado em (LI et al., 2017) porque este executou os experimentos apenas quatro vezes e apresentou a média dos resultados.

Os experimentos foram submetidos para execução sob diferentes perfis de configuração do cluster. Definimos 10 perfis de configuração com variações no número de executores e quantidade de núcleos de CPU e memória RAM por executor. Os 10 perfis de configuração podem ser vistos na Tabela 4.1. Para mostrar a influência no número de executores, as configurações da *conf1* até a *conf4* variam o número de executores em relação ao número de nós no cluster (3). Nessas quatro configurações, o número de núcleos de CPU (8) e quantidade de memória RAM (6 GB) em cada nó foram igualmente divididos pela quantidade de executores em cada nó. Essas quatro configurações são as únicas dentre os 10 perfis que utilizam todos os recursos disponíveis no cluster.

Para mostrar a influência do número de núcleos de CPU, fixamos o número de executores e memória RAM e variamos a quantidade de núcleos disponíveis em cada executor. Essa variação está presente na configuração *conf1*, que foi utilizada como base para os outros perfis, e nas configurações *conf5* à *conf7*. Seguindo o mesmo padrão, fixamos o número de executores e número de núcleos de CPU e variamos a quantidade de memória em cada executor para mostrar a influência da memória RAM na execução. Essa variação pode ser vista na configuração *conf1* e nas configurações *conf8* à *conf10*. Com essas 10 configurações, conseguimos demonstrar diferentes formas de configurar o cluster e a influência individual que cada um dos três parâmetros (número de executores, memória RAM e núcleos de CPU) tem no desempenho da aplicação. Todas as aplicações deste experimento foram executadas utilizando essas 10 configurações. Assim sendo, cada aplicação foi executada oito vezes em cada configuração, totalizando 80 execuções para cada aplicação.

4.2.3 Conjuntos de Dados

Nos experimentos foram utilizados conjuntos de dados de três fontes diferentes. Esses conjuntos foram escolhidos de acordo com o tipo de análise que queríamos fazer de modo a utilizar os recursos de Spark e simular os problemas descritos na taxonomia. Esses conjuntos de dados são apresentados e caracterizados a seguir:

AMPLab Big Data Benchmark: é um *benchmark* para sistemas de análise de grandes volumes de dados desenvolvido no laboratório *AMPLab* na *UC Berkley* (AMPLAB, 2019). Este benchmark foi baseado no apresentado em (PAVLO et al., 2009). Os conjuntos de

Tabela 4.1 – Perfis de configuração do cluster.

Conf ID	Executores	Memória	Núcleos
<i>conf1</i>	3	6 GB	8
<i>conf2</i>	6	3 GB	4
<i>conf3</i>	12	1536 MB	2
<i>conf4</i>	24	768 MB	1
<i>conf5</i>	3	6 GB	1
<i>conf6</i>	3	6 GB	2
<i>conf7</i>	3	6 GB	4
<i>conf8</i>	3	3 GB	8
<i>conf9</i>	3	1536 MB	8
<i>conf10</i>	3	768 MB	8

dados desse benchmark consistem em um conjunto de documentos HTML não estruturados (*Documents*) e duas tabelas estruturadas (*Rankings* e *UserVisits*). Esses conjuntos de dados estão disponíveis em diferentes versões com diferentes escalas de tamanho. Nos nossos experimentos, utilizamos as duas tabelas estruturadas com fator de escala 5. A tabela *Ranking* possui um total de 6.38 GB de dados e a tabela *UserVisits* possui 126.8 GB de dados.

MovieLens: é um sistema para recomendação de filmes² que permite que usuários atribuam notas a filmes e recebam recomendações de outros filmes com base em suas preferências (HARPER; KONSTAN, 2015). O grupo de pesquisa *GroupLens*³ vem coletando e disponibilizando dados do sistema MovieLens desde 1998 para propósitos educacionais, de pesquisa e indústria. Diversas versões dos conjuntos de dados do MovieLens estão disponíveis, com dados coletados em diferentes anos e com diferentes fatores de tamanho. Nos nossos experimentos, utilizamos duas versões. A primeira é o conjunto de dados disponibilizado em 2003 que contem 1 milhão de notas atribuídas por 6000 usuários a 4000 filmes. O segundo conjunto utilizado foi publicado em 2017 e contem 26 milhões de notas atribuídas a 45 mil filmes por 270 mil usuários.

Yelp: é uma rede social⁴ para resenhas e recomendações de restaurantes e hotéis, além de outros tipos de empresas. O Yelp disponibiliza um conjunto de dados aberto com parte dos dados de empresas, resenhas e usuários para propósitos acadêmicos. O conjunto de dados tem 5.2 milhões de resenhas sobre 174 mil empresas, além de outros dados. Nos nossos experimentos, utilizamos o conjunto de dados com as resenhas que possui 3 GB de dados (consideramos apenas o texto da resenha).

² <https://movielens.org>

³ <https://grouplens.org>

⁴ <https://www.yelp.com>

4.2.4 Aplicações

O conjunto de aplicações Spark selecionado para este experimento foi escolhido com o objetivo de mostrar experimentalmente os problemas descritos na taxonomia. Dessa forma, todos os problemas descritos na taxonomia, com exceção dos problemas descritos na primeira categoria (*I1*), possuem ao menos uma aplicação simulando-os. Os problemas da categoria *I1* são relacionados com as configurações do cluster, então os problemas dessa categoria foram simulados através dos diferentes perfis de configuração apresentados anteriormente. Para o desenvolvimento dessas aplicações, utilizamos como referências exemplos utilizados para ilustrar os recursos de Spark e exemplos de aplicações apresentados em (KARAU; WARREN, 2017), assim como aplicações mais específicas apresentadas em (SARWAR et al., 2001) e (AMPLAB, 2019).

Uma vez que queremos demonstrar o impacto dos problemas no desempenho de execução, desenvolvemos diferentes versões das aplicações de modo a ter ao menos uma versão em que o problema está presente e uma versão em que o problema não está presente. Com isso, pudemos comparar os resultados de desempenho de uma aplicação com e sem o problema para avaliar o impacto deste no desempenho da execução. As aplicações são mostradas na Tabela 4.2, onde é possível ver uma descrição, problema relacionado e conjunto de dados utilizado. As aplicações estão disponíveis publicamente no repositório (NETO et al., 2019).

4.2.5 Resultados

Os resultados dos desempenhos de execução das aplicações serão apresentados em termos dos tempos de execução e informações sobre entradas e saídas de dados. Uma vez que os problemas do primeiro grupo (*I1*) são relacionados com configurações do cluster e não com a aplicação, vamos utilizar os resultados agregados de todas as aplicações para analisar os problemas desse grupo. A Tabela 4.3 apresenta os resultados agregados das aplicações para cada perfil de configuração apresentado na Tabela 4.1. Nela é possível ver o número total de *jobs*, estágios e tarefas executados com sucesso ou falhas, além de informações sobre o tempo como a duração da aplicação, tempo de execução acumulado e tempo de CPU. Esses resultados mostram a influência que os parâmetros de configuração do cluster tem no desempenho geral.

Para investigar problemas específicos e comparar seus impactos, analisamos os resultados de desempenho utilizando uma configuração fixa. Escolhemos o perfil de configuração *conf1* como base uma vez que este foi utilizado como referência em todos os parâmetros de configuração. A Tabela 4.4 apresenta os resultados individuais de desempenho de cada uma das 21 aplicações executadas neste experimento, incluindo o número de *jobs*, estágios e tarefas, além dos tempos da aplicação. Os resultados referentes as entradas e saídas de dados é apresentado na Tabela 4.5, incluindo o total de bytes que

Tabela 4.2 – Aplicações executadas nos experimentos.

App ID	Aplicação	Descrição	Problema	Conjunto de Dados	
1	Join Query - Diferentes Particionadores	Essa aplicação realiza uma junção entre dois RDDs que foram particionados por diferentes ou mesmos particionadores.	<i>I5.2</i>	AMPLab Big Data Benchmark	
2	Join Query - Mesmos Particionadores				
3	Aggregation Query - groupByKey	Essa aplicação realiza uma agregação de alta cardinalidade utilizando as transformação <i>groupByKey</i> ou <i>reduceByKey</i> .	<i>I6.1</i>		
4	Aggregation Query - reduceByKey				
5	Distinct User Visits Per Page - aggregateByKey	Essa aplicação realiza uma agregação com mudança de tipo utilizando as transformações <i>aggregateByKey</i> ou <i>reduceByKey</i> .	<i>I6.2</i>		
6	Distinct User Visits Per Page - reduceByKey				
7	Scan and Filter Query - coalesce	Essa aplicação mapeia e filtra o conjunto de dados e, além disso, faz uma mudança no número de partições do RDD utilizando <i>coalesce</i> ou <i>repartition</i> .	<i>I6.3</i>	MovieLens	
8	Scan and Filter Query - repartition				
9	Movies Ratings Average - Utilizando variáveis de transmissão	Essa aplicação calcula as médias das notas dos filmes e salva os resultados utilizando seus nomes. O mapeamento dos identificadores (IDs) dos filmes para os seus nomes é feito com o uso de variáveis que associam IDs com nomes (dicionários). Essas variáveis são transmitidas com e sem variáveis de transmissão.	<i>I3.1</i>		
10	Movies Ratings Average - Sem variáveis de transmissão				
11	Movies Recommendation - 24 Partições	Essa aplicação recomenda filmes utilizando uma técnica de Filtragem Colaborativa baseada em itens. Nessa aplicação, o número de partições do RDD é alterado para múltiplos do número de núcleos de CPU disponíveis no cluster (24, 48, 72 and 96).	<i>I2.1</i>		
12	Movies Recommendation - 48 Partições				
13	Movies Recommendation - 72 Partições				
14	Movies Recommendation - 96 Partições				
15	MovieLens Exploration - Não persistindo	Essa aplicação explora o conjunto de dados do MovieLens fazendo diferentes análises e chamando várias ações. Os RDDs utilizados em mais de uma ação podem ter sido persistidos ou não utilizando diferentes níveis de persistência.	<i>I4.1, I4.2</i>		
16	MovieLens Exploration - Persistindo em memória				
17	MovieLens Exploration - Persistindo em disco				
18	MovieLens Exploration - Persistindo em memória e disco				
19	MovieLens Exploration - Persistindo em memória com serialização dos dados				
20	NGrans Count - Sem Pré-Particionamento	Essa aplicação calcula os n-grams em um conjunto de dados e conta a frequência de cada n-gram. Na aplicação é utilizada a operação <i>reduceByKey</i> e esta foi aplicada em um RDD com e sem pré-particionamento.	<i>I5.1</i>	Yelp	
21	NGrans Count - Com Pré-Particionamento				

Tabela 4.3 – Resultados de desempenho agregados para cada perfil de configuração.

Conf ID	J	FJ	S	FS	T	FT	AD	RT	CT	DT	DCT	RST	JGCT
conf1	92	0	197	0	15084	0	2,1 h	42,4 h	20,4 h	12 min	3,0 min	3 s	9,3 h
conf2	92	0	197	0	15125	2	2,0 h	39,7 h	18,7 h	19 min	4,3 min	3 s	7,4 h
conf3	91	0	208	3	15554	45	2,1 h	39,1 h	16,7 h	27 min	6,9 min	4 s	4,8 h
conf4	89	5	275	30	22154	528	3,1 h	44,8 h	16,9 h	1,0 h	14 min	6 s	3,0 h
conf5	92	0	197	0	15084	0	3,8 h	10,9 h	9,4 h	3,0 min	2,3 min	1,0 s	38 min
conf6	92	0	197	0	15084	0	2,4 h	13,1 h	10,6 h	4,5 min	2,4 min	1 s	1,3 h
conf7	92	0	197	0	15084	0	2,0 h	21,0 h	14,3 h	7,0 min	2,7 min	1 s	3,4 h
conf8	92	0	197	0	15084	0	2,2 h	42,9 h	19,7 h	12 min	3,0 min	2 s	11,9 h
conf9	92	0	197	0	15089	5	2,4 h	47,8 h	19,9 h	13 min	3,0 min	4 s	16,4 h
conf10	83	2	209	15	14989	215	3,9 h	57,4 h	21,6 h	14 min	3,0 min	8 s	24,3 h

Legendas: Conf ID: ID da Configuração; J: Número total de Jobs; FJ: Número total de Jobs que falharam; S: Número total de Estágios; FS: Número total de Estágios que falharam; T: Número total de Tarefas; FT: Número total de Tarefas que falharam; AD: Duração total das Aplicações; RT: Tempo total de execução dos executores; CT: Tempo total de CPU dos executores; DT: Tempo total gasto com desserialização pelos executores; DCT: Tempo total de CPU gasto com desserialização pelos executores; RST: Tempo total gasto com serialização de resultados; JGCT: Tempo total gasto com o processo de Garbage Collection pela JVM.

Tabela 4.4 – Resultados de tempo de desempenho para cada aplicação.

App ID	Jobs	Stages	Tasks	AD	RT	CT	DT	DCT	RST	JGCT
1	2	7	1377	12 min	4,4 h	2,1 h	54 s	9 s	0,2 s	43 min
2	2	6	1335	12 min	4,3 h	2,0 h	59 s	8 s	0,3 s	30 min
3	1	2	2134	9,0 min	3,5 h	1,8 h	49 s	16 s	0,3 s	54 min
4	1	2	2134	8,2 min	3,2 h	1,4 h	49 s	16 s	0,3 s	48 min
5	1	2	2134	10 min	4,0 h	1,8 h	54 s	18 s	0,3 s	44 min
6	1	2	2134	11 min	4,2 h	2,2 h	56 s	17 s	0,3 s	53 min
7	1	1	50	20 s	4,9 min	3,1 min	29 s	4 s	42 ms	16 s
8	1	2	150	59 s	18 min	9,6 min	29 s	4 s	23 ms	46 s
9	2	4	582	38 s	5,8 min	3,4 min	20 s	8 s	74 ms	11 s
10	2	4	582	51 s	4,2 min	2,6 min	39 s	19 s	81 ms	32 s
11	4	9	194	7,1 min	2,3 h	53 min	43 s	7 s	45 ms	52 min
12	4	9	386	4,6 min	1,4 h	42 min	41 s	8 s	0,2 s	28 min
13	4	9	578	3,8 min	1,1 h	32 min	43 s	9 s	0,1 s	17 min
14	4	9	770	3,3 min	56 min	28 min	44 s	10 s	0,2 s	14 min
15	12	25	80	1,0 min	3,8 min	3,4 min	9 s	4 s	12 ms	13 s
16	12	25	80	52 s	3,3 min	2,8 min	9 s	4 s	11 ms	16 s
17	12	25	80	58 s	3,8 min	3,4 min	9 s	4 s	14 ms	11 s
18	12	25	80	53 s	3,4 min	2,9 min	10 s	4 s	13 ms	17 s
19	12	25	80	1,0 min	3,9 min	3,5 min	10 s	4 s	12 ms	12 s
20	1	2	64	29 min	8,5 h	3,7 h	27 s	3 s	5 ms	2,3 h
21	1	2	80	11 min	3,7 h	2,2 h	28 s	4 s	9 ms	33 min

Legendas: App ID: ID da Aplicação; AD: Duração da Aplicação; RT: Tempo total de execução dos executores; CT: Tempo total de CPU dos executores; DT: Tempo total gasto com desserialização pelos executores; DCT: Tempo total de CPU gasto com desserialização pelos executores; RST: Tempo total gasto com serialização de resultados; JGCT: Tempo total gasto com o processo de Garbage Collection pela JVM.

são lidos e escritos em cada aplicação.

4.3 Discussões

Vamos analisar os resultados dos experimentos e discutir o impacto que cada problema descrito na taxonomia tem no desempenho de execução da aplicação. A análise do impacto levará em consideração principalmente o tempo de duração das aplicações. Para essa análise, vamos considerar o *speedup* como métrica de comparação entre as diferentes configurações e versões de uma mesma aplicação. Para o cálculo do *speedup*, escolhemos o tempo de uma configuração ou aplicação como base e dividimos esse pelo tempo de duração das configurações ou aplicações ao qual queremos comparar. Quanto maior o *speedup*, melhor o desempenho em relação à base. Vamos discutir os problemas seguindo as mesmas categorias apresentadas na taxonomia.

I1 - Gerenciamento de Recursos do Cluster: para investigar os problemas relaciona-

Tabela 4.5 – Resultados de entradas e saídas para cada aplicação.

App ID	IR	OW	SR	SW	CM	CD
1	123,0 GB	124,0 MB	92,0 MB	6,0 GB	0 Bytes	0 Bytes
2	123,0 GB	124,0 MB	92,0 MB	6,0 GB	0 Bytes	0 Bytes
3	118,0 GB	6,0 MB	6,0 GB	6,0 GB	0 Bytes	0 Bytes
4	118,0 GB	6,0 MB	1611,0 MB	1611,0 MB	0 Bytes	0 Bytes
5	118,0 GB	15,0 GB	12,0 GB	12,0 GB	0 Bytes	0 Bytes
6	118,0 GB	15,0 GB	12,0 GB	12,0 GB	0 Bytes	0 Bytes
7	5,0 GB	12,0 MB	0 Bytes	0 Bytes	0 Bytes	0 Bytes
8	5,0 GB	21,0 MB	5,0 GB	5,0 GB	0 Bytes	0 Bytes
9	676,0 MB	1877,0 KB	1702,0 KB	136,0 MB	0 Bytes	0 Bytes
10	676,0 MB	1877,0 KB	1706,0 KB	136,0 MB	0 Bytes	0 Bytes
11	23,0 MB	101,0 MB	63,0 KB	1865,0 MB	0 Bytes	0 Bytes
12	23,0 MB	156,0 MB	63,0 KB	1949,0 MB	0 Bytes	0 Bytes
13	23,0 MB	159,0 MB	64,0 KB	1999,0 MB	0 Bytes	0 Bytes
14	23,0 MB	169,0 MB	64,0 KB	2034,0 MB	0 Bytes	0 Bytes
15	711,0 MB	4,0 MB	2,0 MB	467,0 MB	0 Bytes	0 Bytes
16	779,0 MB	4,0 MB	0 Bytes	467,0 MB	1144,0 MB	0 Bytes
17	712,0 MB	4,0 MB	0 Bytes	467,0 MB	0 Bytes	262,0 MB
18	779,0 MB	4,0 MB	0 Bytes	467,0 MB	1144,0 MB	0 Bytes
19	712,0 MB	4,0 MB	0 Bytes	467,0 MB	262,0 MB	0 Bytes
20	3,0 GB	3,0 GB	3,0 GB	3,0 GB	0 Bytes	0 Bytes
21	3,0 GB	3,0 GB	7,0 GB	7,0 GB	0 Bytes	0 Bytes

Legendas: App ID: ID da Aplicação; IR: Total de bytes de entrada lidos; OW: Total de bytes de saída escritos para o HDFS; SR: Total de bytes lidos no processo de distribuição (shuffle); SW: Total de bytes escritos no processo de distribuição (shuffle); CM: Total de bytes persistidos em memória; CD: Total de bytes persistidos em disco.

dos com as configurações do cluster, nós analisamos a influência dos três parâmetros que normalmente são escolhidos quando uma aplicação é submetida para execução: número de executores e quantidade de núcleos de CPU e memória RAM para cada executor. A Figura 4.10 apresenta os speedups do tempo total de duração de todas as aplicações em cada um dos perfis de configuração, tendo como base o tempo da configuração *conf1* que foi utilizado como referência em todos os outros perfis.

Primeiro vamos analisar o impacto da variação do número de executores no desempenho, que pode ser observado nos *speedups* das configurações *conf1*, *conf2*, *conf3*, e *conf4*. Podemos observar que as configurações *conf2* e *conf3*, que tinham de duas a quatro vezes o número de executores de *conf1*, respectivamente, obtiveram um resultado de 6% a 3% melhor que *conf1*, mesmo tendo um maior número de falhas em jobs, estágios e tarefas, como observado na Tabela 4.3. Dentre as quatro configurações, a que apresenta a maior discrepância em comparação as outras é a *conf4*, que era a configuração com o maior número de executores e menor número de recursos por executor. Essa configuração

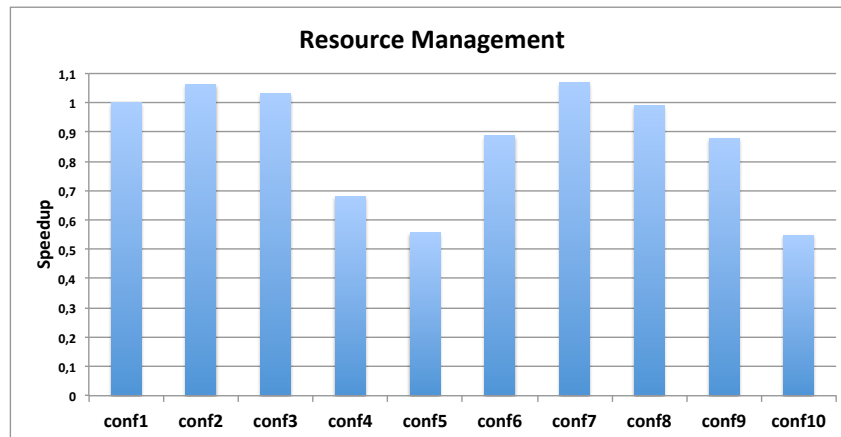


Figura 4.10 – *Speedups* dos experimentos de Gerenciamento de Recursos do Cluster.

teve um desempenho 32% pior que a *conf1* e o maior número de falhas dentre todas as configurações.

A influência do número de núcleos de CPU pode ser vista nos resultados de desempenho das configurações *conf5*, *conf6*, *conf7* e a *conf1*. Essas configurações possuem um número fixo de executores e de memória RAM por executor e variam no número de núcleos de CPU. Esse número impacta na quantidade de tarefas que podem ser executadas em paralelo. Na Figura 4.10 podemos observar que a configuração *conf7* obteve um resultado 7% melhor que a *conf1* mesmo tendo utilizado metade do número de núcleos de CPU que este. Utilizando um quarto do número de núcleos disponíveis, a *conf6* teve um desempenho 12% pior em relação ao *conf1*. A configuração *conf5* que teve o menor número de núcleos de CPU por executor teve o pior desempenho dentre as quatro configurações. Esses resultados mostram que o número de núcleos de CPU tem um impacto direto no desempenho de execução. Apesar disso, ter todos os núcleos disponíveis não significa ter o melhor desempenho, como os resultados de *conf1* mostraram.

O impacto que a quantidade de memória RAM tem no desempenho de execução pode ser observado nos resultados das configurações *conf1*, *conf8*, *conf9* e *conf10*. Essas quatro configurações possuem um número fixo de executores e de núcleos de CPU por executor, mas variam na quantidade de memória RAM. Utilizando metade da memória disponível, a configuração *conf8* teve um desempenho similar *conf1* que utilizava toda a memória, variando apenas 1% para pior. Os resultados obtidos com as configurações *conf9* e *conf10* mostraram que reduzir ainda mais a quantidade de memória piora o desempenho de execução. Essas configurações possuíam 1/4 e 1/8 da memória disponível, respectivamente, e obtiveram desempenhos 12% e 46% pior que *conf1*. Além disso, essas configurações tiveram falhas que foram relacionadas a falta de memória que não foram observadas nas configurações *conf1* e *conf8*.

Os resultados de desempenho acumulado para cada configuração corroboram os problemas descritos na categoria de *Gerenciamento de Recursos do Cluster*. As configura-

ções que possuíam a menor quantidade de memória por executor (*conf4* e *conf10*) foram as que tiveram o maior número de falhas e os piores desempenhos. Especialmente no caso da configuração *conf10*, que tinha um número fixo de executores e núcleos, foi possível observar que uma pouca quantidade de memória pode ser responsável por uma queda de desempenho e um maior número de falhas na aplicação, como descrito no problema *I1.1*. No geral, todas as configurações sustentam o que é descrito no problema *I1.2* uma vez que a maneira que o cluster é configurado tem impacto direto no desempenho da aplicação.

I2 - Particionamento dos Dados: para demonstrar a influência do número de partições de um RDD no desempenho da aplicação, como discutido no problema *I2.1*, executamos uma aplicação em que a quantidade de partições do RDD foi dividida em diferentes números. A aplicação implementava um sistema de recomendação de filmes utilizando uma técnica de Filtragem Colaborativa baseada em itens (SARWAR et al., 2001). O número de partições foi variado nas aplicações de acordo com o número de núcleos de CPU disponíveis (24), possuindo de uma a quatro vezes essa quantidade de partições. Essas aplicações são as com identificador 11, 12, 13 e 14, como apresentado na Tabela 4.2. Os seus resultados de desempenho podem ser vistos em seus respectivos identificadores na Tabela 4.4.

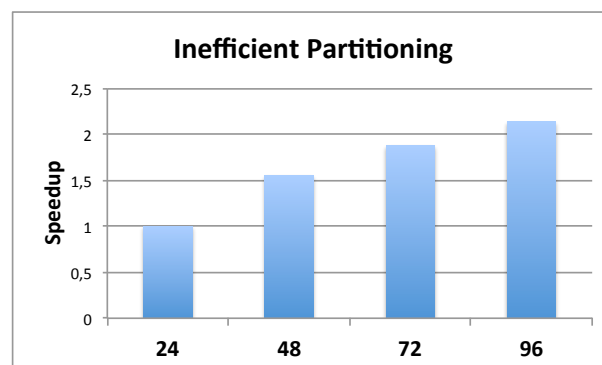


Figura 4.11 – Speedups dos experimentos de Particionamento de Dados.

A Figura 4.11 apresenta os speedups dessas aplicações tendo como base a aplicação 11 que utilizou 24 partições. Podemos observar que o desempenho de execução foi melhor na medida em que o número de partições cresceu. A aplicação 12, que tinha 48 partições, teve um resultado 55% melhor que a aplicação 11. Já a aplicação com o maior número de partições (96), teve um resultado mais que duas vezes melhor, o que significa que nos nossos experimentos ela foi executada em menos da metade do tempo que a aplicação 11. Ao comparar a diferença de desempenho entre as quatro, podemos ver que a diferença entre duas aplicações sucessivas (a com 48 e 72 partições, por exemplo), diminui na medida em que o número de partições vai subindo. Esse resultado sugere que ao aumentar ainda mais o número de partições, é possível chegar a um limite de partições que quando ultrapassado o desempenho de execução começa a cair ao invés de melhorar.

Nesse experimento nós não quisemos identificar esse limite, mas mostrar a influência que o número de partições de um RDD pode ter no desempenho da aplicação. Dessa forma, esses resultados mostram o impacto da estratégia de particionamento que é descrito no problema *I2.1*.

I3 - Transmissão de Dados: desenvolvemos uma aplicação que calcula a média das notas recebidas por um filme no conjunto de dados do *MovieLens* para investigar o problema *I3.1*. Nessa aplicação, temos um conjunto de dados com os identificadores (IDs) dos filmes e suas notas e um outro conjunto de dados contendo os seus IDs e nomes. Uma vez que a quantidade de dados no conjunto com nomes é muito menor que o do conjunto de notas, combinamos as duas tabelas fazendo uso de uma variável *dicionário* que associa IDs com nomes. Para ver a influência de variáveis de transmissão no desempenho, desenvolvemos uma versão em que a variável é referenciada utilizando uma variável de transmissão (Aplicação 9) e uma versão sem utilizar variável de transmissão (Aplicação 10). Os resultados de ambas podem ser vistas na Tabela 4.4.

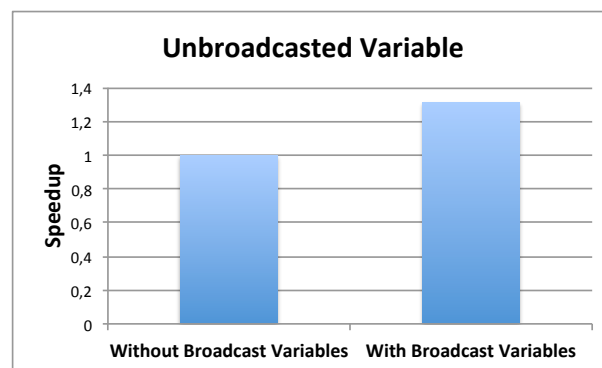


Figura 4.12 – Speedups dos experimentos de Transmissão de Dados.

A Figura 4.12 apresenta os speedups para os dois casos tomando como base a aplicação que não utilizou variáveis de transmissão. Podemos ver que a aplicação que utilizou variáveis de transmissão obteve um desempenho 31% melhor que do que a versão que não utilizou. Esse resultado confirma o que é descrito no problema *I3.1* uma vez que mostra que não utilizar variáveis de transmissão em ocasiões que a variável tem um grande tamanho ou é copiada muitas vezes causa uma perda de desempenho na aplicação.

I4 - Persistência de Dados: para investigar os problemas relacionados com a persistência de dados desenvolvemos uma aplicação que explora o conjunto de dados do *MovieLens*. Essa aplicação obtém diferentes informações como o número de filmes produzidos por ano, os gêneros de filmes mais populares por ano e os filmes mais bem avaliados por década, além de outras informações. Para cada informação obtida, uma ação é chamada para salvar os dados obtidos no HDFS e outra para exibir os resultados no programa Driver. Dessa forma, diferentes RDDs são utilizados em mais de uma ação. Com o intuito de

avaliar a influência da persistência de RDDs e o nível de persistência, executamos diferentes versões dessa aplicação no qual tem uma versão em que nenhum RDD reutilizado foi persistido e outras em que os RDDs foram persistidos utilizando diferentes níveis de persistência. Os níveis de persistência utilizados foram: em memória (*MEMORY_ONLY*), em que o RDD ou parte dele é salvo na memória e a parte que não cabe é recomputado quando necessário; em disco (*DISK_ONLY*), em que o RDD é salvo no disco; em memória e disco (*MEMORY_AND_DISK*), em que parte do RDD é salvo em memória e a parte que não cabe é salva em disco; e em memória com os dados serializados (*MEMORY_ONLY_SER*), no qual os dados do RDD são serializados antes de serem salvos na memória.

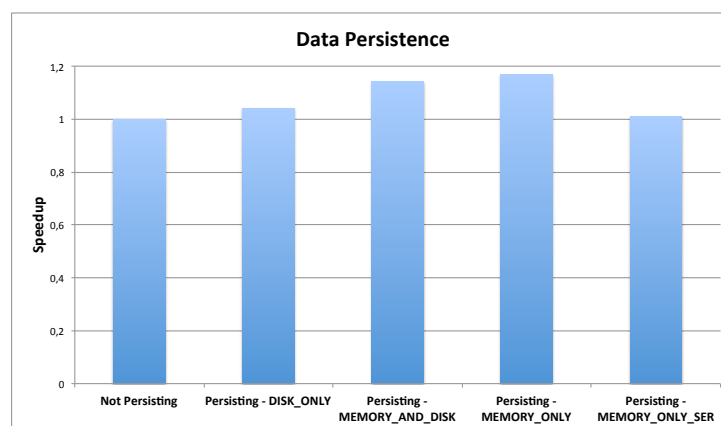


Figura 4.13 – Speedups dos experimentos de Persistência de Dados.

As aplicações 15 a 19 da Tabela 4.2 são as que tratam da persistência de dados. Os resultados do seu desempenho em relação ao tempo podem ser vistos na Tabela 4.4 e os resultados sobre entradas e saídas podem ser vistos na Tabela 4.5, onde é possível ver a quantidade de dados (bytes) salvos em memória e em disco. A Figura 4.13 apresenta os speedups desses experimentos tendo como base a aplicação que não persistiu os RDDs. Podemos ver a influência de persistir os dados ou não ao comparar a aplicação que não persistiu (Aplicação 15) com as demais. Com exceção da aplicação que persistiu em memória com serialização dos dados (Aplicação 19), que teve um desempenho semelhante, as demais aplicações tiveram um desempenho melhor. No caso com serialização, o tempo gasto com essa serialização dos dados reduziu o tempo ganho com a persistência, o que justifica a semelhança. Já os outros casos obtiveram desempenhos de 4% à 17% melhor, o que mostra que persistir RDDs que são reutilizados melhora o desempenho da aplicação.

O problema *I4.2* fala sobre a importância do nível de persistência no desempenho da aplicação. Podemos ver essa influência ao comparar os resultados das aplicações que persistiram os dados. Podemos ver que os melhores resultados foram obtidos com as aplicações que persistiram em memória (Aplicação 16) e em memória e disco (Aplicação 18). Ambas tiveram um resultado semelhante porque a quantidade de dados persistidos foram suficientes para caber na memória, não sendo necessário recomputar ou salvar em

disco parte dos dados, como podemos ver na Tabela 4.5. Persistir os dados em disco tem a sobrecarga causada pela leitura e escrita de dados em disco que é mais lento que em memória. Apesar disso, persistir os dados ou parte deles em disco pode ser uma boa opção em casos em que existe uma grande quantidade de dados para serem persistidos e estes não cabem na memória, o que não foi o caso do nosso experimento. O caso que teve o pior desempenho foi a aplicação em que os dados foram persistidos em memória com serialização dos dados. Mesmo assim, a quantidade de dados salvos na memória foi menor nesse caso, como pode ser visto na Tabela 4.5, algo que pode ser uma melhor opção nos casos em que a memória fica cheia. Esses resultados mostram a influência do nível de persistência no desempenho de uma aplicação, como descrito no problema *I4.2*.

I5 - Redistribuição de Dados (Data Shuffling): para investigar o problema *I5.1* escolhemos uma aplicação que calcula *n-grams* e suas frequências em um conjunto de dados. *N-grams* são sequências contíguas de *n* palavras coletadas a partir de um texto. Desenvolvemos duas versões dessa aplicação, uma em que o RDD de chaves/valor não é pré-particionado antes da operação *reduceByKey* (Aplicação 20) ser chamada e uma versão em que o RDD é particionado antes dessa operação (Aplicação 21). O desempenho de ambas pode ser visto na Tabela 4.4. Os speedups dessas aplicações tendo como base a aplicação que não pré-particionou pode ser visto na Figura 4.14. Podemos ver que o desempenho da versão em que o RDD foi pré-particionado foi mais que duas vezes melhor que a versão que não particionou. Esse resultado corrobora o que é descrito no problema *I5.1* em que pré-particionar ou não um RDD do tipo chave/valor antes de operações que envolvem redistribuição pode ter um grande impacto no desempenho da aplicação.

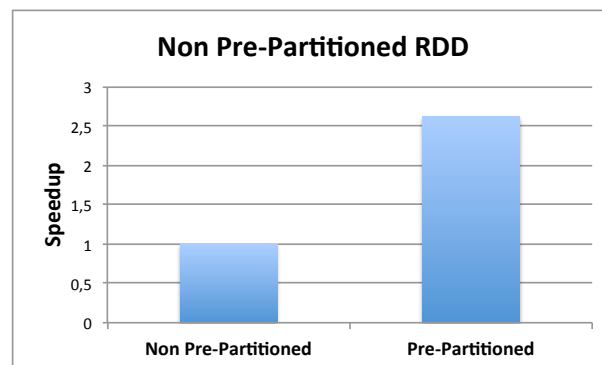


Figura 4.14 – Speedups dos experimentos de RDD Não Pré-Particionado.

Para os experimentos relacionados ao problema *I5.2* desenvolvemos uma aplicação que realiza uma junção entre dois RDD baseado no *Join Query* do benchmark do *AMPLab Big Data Benchmark* (AMPLAB, 2019). Para ver a influência de fazer uma junção com RDDs co-particionados ou não, executamos uma versão em que ambos os RDDs são pré-particionados utilizando diferentes particionadores (Aplicação 1) e uma versão em que utilizamos os mesmos particionadores para os dois (Aplicação 2). Os seus resultados

de desempenho podem ser vistos na Tabela 4.4 e os speedups tendo como base o caso em que os RDDs não são co-particionados podem ser vistos na Figura 4.15. É possível ver que ambos os casos tiveram um desempenho muito semelhante, variando apenas 1% para melhor no caso em que os RDDs são co-particionados.

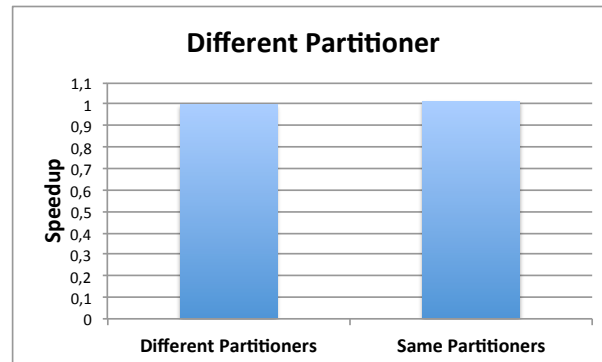


Figura 4.15 – Speedups dos experimentos de Particionamentos Diferentes.

Apesar da pouca diferença no desempenho, é possível observar outras diferenças entre as duas aplicações, como o número de estágios e tarefas. Na Tabela 4.4 é possível ver que a aplicação que não tinha os RDDs co-particionados (Aplicação 1) teve um estágio de redistribuição a mais que a outra com co-particionamento (Aplicação 2), além de um maior número de tarefas. Esse resultado corresponde com o que é esperado na teoria e descrito no problema *I5.2* uma vez que o caso com co-particionamento reduziu a redistribuição de dados. Entretanto, nossos experimentos não apresentaram uma diferença significativa nos desempenhos. Isso pode ter ocorrido devido as nossas escolhas no desenvolvimento da aplicação, como o desenvolvimento dos particionadores, ou devido ao nosso ambiente de cluster limitado. Assim sendo, consideramos que mais experimentos são necessários para investigar esse problema, esperamos que diferenças mais notáveis possam ser observadas através de mudanças na aplicação e em um ambiente de cluster com mais recursos.

I6 - Projeto da Aplicação: para investigar o problema *I6.1* que trata do impacto de escolher *groupByKey* ao invés de *reduceByKey* para aplicar uma função a valores associados a uma chave, desenvolvemos uma aplicação baseada na *Aggregation Query* do benchmark *AMPLab Big Data Benchmark* (AMPLAB, 2019). Essa aplicação faz uma agregação para somar todos os valores associados a uma mesma chave. Desenvolvemos uma versão em que essa agregação é feita utilizando a operação *groupByKey* (Aplicação 3) e outra em que é utilizado a operação *reduceByKey* (Aplicação 4). O desempenho de ambos podem ser vistos na Tabela 4.4 e os seus speedups normalizados pelo caso que utilizou *groupByKey* podem ser vistos na Figura 4.16. Podemos ver que a aplicação que utilizou *reduceByKey* teve um desempenho 10% melhor que a versão que utilizou *groupByKey*. Esse resultado está de acordo com o que é descrito no problema *I5.1* uma vez que observamos um melhor desempenho da aplicação que usou *reduceByKey* devido à forma que essa aplicação

gerencia a redistribuição dos dados que transfere uma quantidade menor de dados pelo cluster.

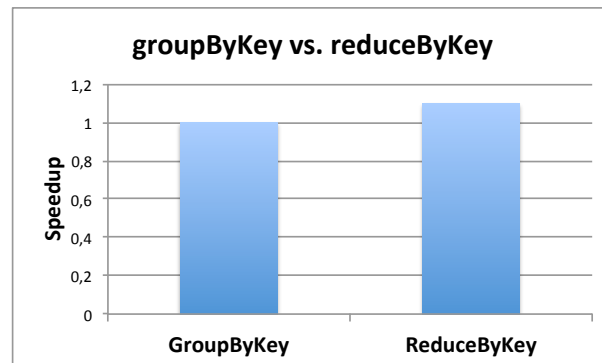


Figura 4.16 – Speedups dos experimentos de *groupByKey* versus *reduceByKey*.

Desenvolvemos uma aplicação que explora o conjunto de dados do *AMPLab Big Data Benchmark* para investigar o problema *I6.2*. A aplicação computa o conjunto de usuários distintos que visitaram um site na tabela *UserVistis*. Para isso, a aplicação faz uma agregação que envolve a mudança do tipo do valor agregado semelhante aos exemplos apresentados nas figuras 4.7 e 4.8. Desenvolvemos uma versão que utiliza a operação *reduceByKey* (Aplicação 6) e outra que utiliza *aggregateByKey* (Aplicação 5). Seus resultados de desempenho podem ser vistos na Tabela 4.4 e seus speedups tendo como base a aplicação que utilizou *reduceByKey* podem ser vistos na Figura 4.17. Nela é possível ver que a aplicação que utilizou *aggregateByKey* teve um desempenho 5% melhor que a aplicação que usou *reduceByKey*. Mesmo não tendo uma diferença muito significativa, conseguimos observar a diferença das duas aplicações em agregações que possuem mudança de tipo, como descrito no problema *I6.2*.

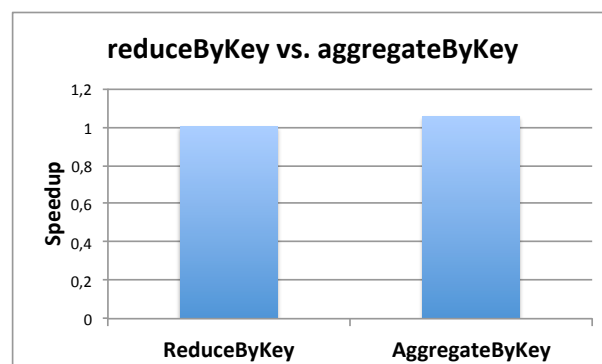


Figura 4.17 – Speedups dos experimentos de *reduceByKey* versus *aggregateByKey*.

Para investigar o problema *I6.3* desenvolvemos uma aplicação baseada no *Scan Query* do benchmark *AMPLab Big Data Benchmark* (AMPLAB, 2019). A aplicação faz uma simples filtragem e seleção na tabela *Rankings*. Para ver a diferença entre as operações *repartition* e *coalesce*, desenvolvemos duas versões dessa aplicação, uma em que reduzimos o número de partições do RDD utilizando *repartition* (Aplicação 8) e outra

que utilizou *coalesce* (Aplicação 7). Seus resultados podem ser vistos na Tabela 4.4 e seus speedups tendo como base a aplicação que usou *repartition* na Figura 4.18. Nela podemos observar uma diferença significativa no desempenho das duas aplicações. A aplicação que utilizou *coalesce* teve um desempenho quase três vezes melhor que a aplicação que utilizou *repartition*. Outra diferença que pode ser observada é no número de estágios na Tabela 4.4. Uma vez que a operação *coalesce* une partições em um mesmo nó ao invés de redistribuir os dados, nos casos de redução de partições, ela consegue evitar um estágio de redistribuição, ao contrário da operação *repartition*. Esses resultados corroboram o que é descrito no problema *I6.3*.

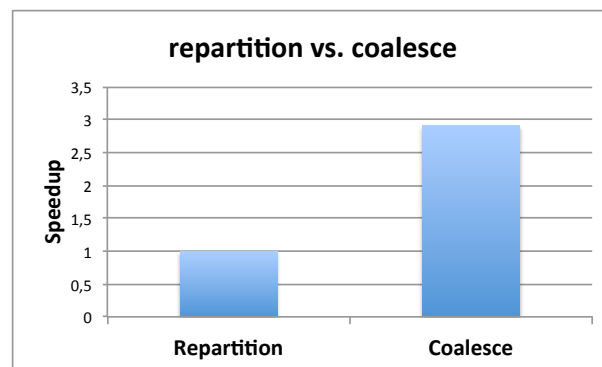


Figura 4.18 – Speedups dos experimentos de *repartition* versus *coalesce*.

De maneira geral, nossos experimentos mostraram que os problemas descritos na taxonomia podem ser observados na prática. É importante destacar que o ambiente de cluster utilizado possui uma quantidade limitada de recursos. Essa situação pode ter impactado na diferença de desempenho entre as aplicações mais eficientes e aquelas com piores resultados. Esperamos que diferenças maiores possam ser observadas em ambientes que possuem uma quantidade maior de recursos e permitem mais possibilidades de configuração.

As maiores diferenças entre speedups foram observadas nos experimentos dos problemas *I5.1* e *I6.3*. Nelas, as versões sem os problemas tiveram um desempenho de 2.6 a 2.9 vezes melhor, respectivamente, que as versões com os problemas. Ambos os problemas eram relacionados com a possibilidade de reduzir ou evitar a redistribuição de dados. Isso mostra que a redistribuição de dados pode ter uma grande influência no desempenho de execução da aplicação e que deve ser levado em conta no seu desenvolvimento. As menores diferenças entre os speedups foram observadas nos experimentos dos problemas *I5.2* e *I6.2*, que tiveram diferenças de apenas 1% e 5%, respectivamente, entre as versões com e sem problemas. Essas pequenas diferenças podem ser justificadas por nossas escolhas nos conjuntos de dados e aplicações. Mesmo assim, pudemos observar que parte dos problemas descritos foi confirmado ao analisar mais resultados além do tempo de duração, como o número de estágios e tarefas. Planejamos revisitar os experimentos desses problemas ao

procurar exemplos e conjuntos de dados em que diferenças nos desempenhos de execução possam ser mais notáveis.

5 Uma Taxonomia de Defeitos Funcionais para Apache Spark

Este capítulo apresenta o resultado de um estudo sobre defeitos em aplicações Spark. A partir desse estudo, foi desenvolvida uma taxonomia que agrupa e categoriza defeitos que podem aparecer no código de aplicações Spark e resultar em falhas na execução. Os defeitos identificados nesta taxonomia surgiram, principalmente, de uma análise empírica da documentação de Spark (SPARK, 2019) e código fonte de aplicações Spark desenvolvidos pelo autor ou outras fontes, como as aplicações apresentadas em (ZAHARIA et al., 2015), (GANELIN et al., 2016) e (KARAU; WARREN, 2017). O objetivo desta taxonomia é modelar possíveis defeitos que podem aparecer no desenvolvimento de aplicações Spark e servir como um guia para o desenvolvimento de aplicações de modo a evitar esses tipos de defeitos, assim como para o desenvolvimento de casos de testes e técnicas de teste para sistemas do tipo.

Este capítulo está organizado da seguinte forma: a Seção 5.1 apresenta a taxonomia de defeitos funcionais para Spark; a Seção 5.2 apresenta uma série de exemplos de aplicações Spark que são utilizados para ilustrar os defeitos descritos na taxonomia; e a Seção 5.3 finaliza o capítulo com discussões sobre a taxonomia.

5.1 Taxonomia

A taxonomia é dividida em três categorias principais, seguindo o tipo de operação ou recurso de Spark: *Transformação*, *Ação* e *Acumuladores*. Além disso, existem subcategorias que visam restringir os defeitos em categorias mais específicas seguindo o tipo de operação que está sendo feita. Alguns dos defeitos identificados foram recorrentes e podem aparecer em operações das três categorias principais, como defeitos relacionados a operações de agregação. Por esse motivo, optamos por repetir os defeitos nas três categorias uma vez que o contexto em que cada uma delas representa é diferente. A taxonomia é apresentada na Figura 5.1. As categorias e defeitos descritos são apresentados a seguir.

F1 - Transformação: este grupo descreve defeitos que podem surgir devido ao uso de operações de transformação em aplicações Spark. Esses defeitos podem ser causados devido a escolhas erradas de programação ou enganos ao utilizar uma transformação em uma aplicação. Por exemplo, o defeito pode ocorrer devido à chamada de uma transformação incorreta ou devido à passagem de um parâmetro errado para a transformação. Defeitos e subgrupos dessa categoria são apresentados a seguir.

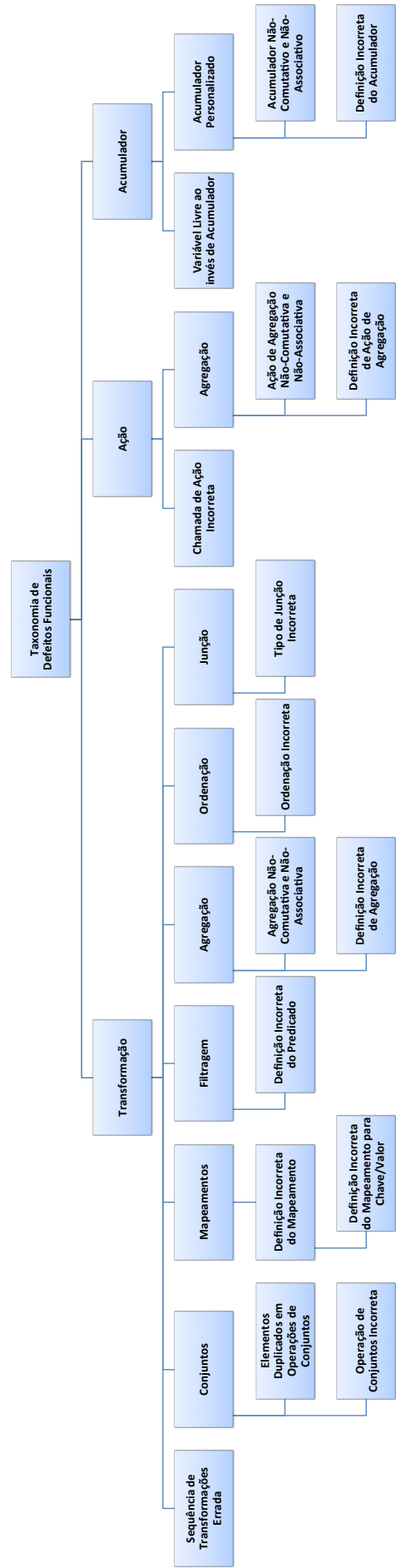


Figura 5.1 – Taxonomia de Defeitos Funcionais.

F1.1: Sequência de Transformações Errada: considerando uma aplicação Spark como uma sequência de transformações em RDDs seguidas por uma ação, este defeito se refere à chamada de uma transformação em uma ordem incorreta na sequência, respeitando os tipos de RDDs de entrada e saída. Por exemplo, em uma sequência de transformações, duas transformações podem ter como entrada e saída RDDs do mesmo tipo. Dessa forma, se a ordem dessas duas transformações forem trocadas uma com a outra na aplicação, esta não irá sofrer um erro de tipagem. Entretanto, a aplicação pode ter um resultado inesperado uma vez que a ordem em que as transformações são feitas é importante na análise.

F1.2: Conjuntos: este subgrupo representa defeitos relacionados com transformações que simulam as operações de conjuntos matemáticos em Spark (*union*, *subtract* e *intersection*) entre dois RDDs. Essas operações possuem comportamentos semelhantes às suas contrapartes em conjuntos matemáticos, entretanto, não respeitam as mesmas propriedades matemáticas.

F1.2.1: Elementos Duplicados em Operações de Conjuntos: apesar de Spark fornecer operações semelhantes às operações de conjuntos matemáticos, nem todas as operações respeitam a propriedade de que não se deve ter elementos duplicados, como ocorre com conjuntos matemáticos. Dessa forma, este defeito se refere ao erro semântico de assumir essa propriedade ao utilizar as operações de conjuntos de Spark. A operação de interseção (*intersection*) é a única das três operações em que o RDD resultante não possui elementos duplicados. Já as operações de união (*union*) e diferença (*subtract*) podem gerar RDDs com elementos duplicados, sendo necessário chamar a transformação *distinct* após elas para retirar elementos duplicados no RDD, quando necessário.

F1.2.2: Operação de Conjuntos Incorreta: este defeito se refere à aplicação de uma transformação de conjuntos incorreta (*union*, *subtract* e *intersection*). Por aplicação incorreta, nos referimos ao engano de fazer a chamada a uma transformação no lugar de outra. Por exemplo, fazer a chamada da transformação de interseção (*intersection*) entre dois RDDs no lugar da transformação de diferença (*subtract*).

F1.3: Mapeamento: defeitos desse grupo são relacionados com as transformações de mapeamento de Spark (*map*, *flatMap*, *mapPartitions*, *mapPartitionsWithIndex*). Essas transformações aplicam funções de transformação passadas como parâmetro aos elementos do RDD de forma a criar um novo RDD que contem os valores mapeados do primeiro RDD.

F1.3.1: Definição Incorreta do Mapeamento: este defeito se refere à definição incorreta de uma função de transformação passada para uma transformação de mapeamento de forma que os elementos do primeiro RDD são mapeados de forma errada para elementos do novo RDD.

F1.3.1.1: Definição Incorreta do Mapeamento para Chave/Valor: este defeito pode ser

considerado como uma especialização do defeito *F1.3.1* pois se refere ao mapeamento incorreto de elementos de um RDD que não possui tuplas de chave/valor para um elemento do tipo chave/valor. Este defeito pode estar relacionado à definição incorreta da chave, valor ou ambos. Consideramos esse defeito de forma separada dado a importância que RDDs do tipo chave/valor têm em Spark, tendo um conjunto próprio de transformações que operam sobre esse tipo.

F1.4: Filtragem: este grupo representa defeitos relacionados com transformações de filtragem de Spark que recebem uma função de predicado como parâmetro e removem elementos que não são avaliados como verdadeiros pela função.

F1.4.1: Definição Incorreta do Predicado: este defeito se refere a definição incorreta da função de predicado utilizada para filtrar elementos em uma transformação de filtragem.

F1.5: Agregação: este grupo representa defeitos relacionados com transformações de agregação que recebem funções de agregação como parâmetro e aplicam essas funções sobre valores agregados por chave em RDDs do tipo chave/valor.

F1.5.1: Agregação Não-Comutativa e Não-Associativa: este defeito se refere à definição incorreta da função de agregação passada para transformações de agregação de modo que a função não respeita as propriedades de comutatividade e associatividade. Uma vez que a função de agregação é aplicada de forma paralela, se ela não for comutativa e associativa, o seu resultado é não determinístico (CHEN et al., 2017).

F1.5.2: Definição Incorreta de Agregação: este defeito se refere à definição incorreta da função de agregação utilizada para agregar valores em transformações de agregação.

F1.6: Ordenação: este grupo representa defeitos relacionados com transformações de ordenação que ordenam RDDs com base em uma chave.

F1.6.1: Ordenação Incorreta: este defeito se refere à incorreta aplicação da transformação de ordenação, como por exemplo ordenar de forma decrescente ao invés de ordem crescente.

F1.7: Junção: este grupo representa defeitos relacionados com transformações de junção que fazem uma junção relacional entre dois RDDs.

F1.7.1: Tipo de Junção Incorreta: este defeito se refere à aplicação de uma transformação de junção incorreta (*join*, *rightOuterJoin*, *leftOuterJoin*, *fullOuterJoin*) de modo que foi escolhida uma junção de um tipo ao invés de outro.

F2 - Ação: este grupo descreve defeitos que podem surgir devido ao uso de operações de ação em aplicações Spark. Assim como os defeitos em transformações, defeitos em ações podem ser causados devido a escolhas erradas de programação ou enganos ao utilizar uma

ação em uma aplicação. Por exemplo, um defeito pode aparecer devido a uma chamada incorreta de uma ação ou a um valor incorreto passado como parâmetro para a ação.

F2.1: Chamada de Ação Incorreta: este defeito se refere à chamada de uma ação incorreta, ou seja, chamar uma ação ao invés de outra. Por exemplo, chamar uma ação relacionada com RDDs de tipos numéricos (DoubleRDDFunctions) incorreta, como a ação *min* ao invés de *max*, entre outras.

F2.2: Agregação: este grupo representa defeitos relacionados com ações de agregação que recebem funções de agregação como parâmetro e aplicam essas funções sob valores de um RDD.

F2.2.1: Ação de Agregação Não-Comutativa e Não-Associativa: de forma semelhante à *F1.5.1*, ações de agregação, como a *reduce* por exemplo, também são aplicadas de forma paralela. Dessa forma, se a função de agregação passada como parâmetro não for comutativa e associativa, seu resultado é não determinístico.

F2.2.2: Definição Incorreta de Ação de Agregação: este defeito se refere à definição incorreta da função de agregação utilizada para agregar valores em ações de agregação.

F3 - Acumuladores: este grupo descreve defeitos relacionados com as variáveis compartilháveis do tipo acumulador. Por exemplo, o defeito pode ocorrer ao se definir um acumulador personalizado.

F3.1: Variável Livre ao invés de Acumulador: este defeito se refere ao uso de uma variável livre (variável utilizada em uma função que foi definida fora de seu escopo) dentro de uma transformação ou ação para propagar valores agregados dos executores para o programa Driver. Quando uma clausura (função que possui referências à variáveis livres) é passada para uma operação Spark, as variáveis livres referenciadas nessa função são copiadas e enviadas para cada tarefa que vai ser executada no cluster. Então, quando uma variável dessas é modificada dentro da operação Spark, seu valor não é propagado para o programa Driver porque foi a variável local que foi modificada e não a que se encontra no Driver. Para que esse valor fosse propagado para o programa Driver, deve-se utilizar um acumulador. Logo, não utilizar um acumulador nesse caso pode ocasionar um comportamento incorreto.

F3.2: Acumulador Personalizado: defeitos desse grupo se referem à definição de um acumulador personalizado. Spark oferece implementações para acumuladores de tipos padrões, como tipos numéricos, por exemplo. Para outros tipos de dados é possível desenvolver um acumulador próprio.

F3.2.1: Acumulador Não-Comutativo e Não-Associativo: este defeito se refere à definição de um acumulador personalizado que não respeita as propriedades de comutatividade e

associatividade. Assim como transformações e ações de agregação, acumuladores agregam valores de forma paralela, logo, se estes não respeitarem essas propriedades o seu resultado é não determinístico.

F3.2.2: Definição Incorreta do Acumulador: este defeito se refere à definição de um acumulador personalizado de forma incorreta.

5.2 Exemplos

Para ilustrar a forma em que o defeitos funcionais descritos na taxonomia podem aparecer no código da aplicação Spark, vamos apresentar uma série de exemplos de códigos de aplicações Spark e, em seguida, apresentar possíveis defeitos de acordo com a taxonomia. Os exemplos serão apresentados a seguir. A Figura 5.2 apresenta o exemplo de uma aplicação Spark que faz uma simples análise em arquivos de *Log*. Um RDD com o conteúdo dos arquivos de *log* é criado na linha 1. Na linha 2, é aplicada uma transformação de filtragem para pegar *logs* de erros. Em seguida, é aplicada uma transformação de mapeamento para pegar apenas uma parte do conteúdo do *log* na linha 3. Na linha 4, é feita uma nova filtragem para limitar o conteúdo que possui uma palavra específica. Por último, é aplicada a ação *count* que conta a quantidade de elementos do RDD.

```
1 val fooCount = sc.textFile("hdfs://logs-path")
2   .filter(_.startsWith("ERROR"))
3   .map(_.split('\\\\t')(2))
4   .filter(_.contains("foo"))
5   .count
```

Figura 5.2 – Exemplo de aplicação Spark que faz uma análise em arquivos de *Log*.

A Figura 5.3 apresenta o exemplo de uma aplicação Spark que computa um valor estimado para a constante π . Este exemplo é um dos exemplos padrões encontrados na documentação de Spark (SPARK, 2019). Esta aplicação estima o valor de π através do *Método de Monte Carlo* ao selecionar pontos aleatórios em um quadrado unitário de $(0, 0)$ à $(1, 1)$ e ver quanto desses pontos caem dentro de um círculo unitário, a fração deste valor deve ser aproximadamente $\pi/4$. Na linha 1, é criado um RDD que contem os números de 1 à n (que representa o número de pontos aleatórios que serão criados). Entre as linhas 2 e 6, é feito um mapeamento no RDD para gerar um ponto aleatório e verificar se este está (1) ou não (0) dentro do círculo. A contagem de pontos dentro do círculo é feita na linha 7 através da chamada da ação *reduce* que soma todos os valores do RDD e o valor estimado de π é impresso na linha 8.

A Figura 5.4 apresenta o exemplo de uma aplicação Spark que computa *n-grams* a partir de um conjunto de dados de textos e depois calcula a frequência de cada *n-gram*. Este exemplo tem a mesma implementação da aplicação 20 (Tabela 4.2) que foi utilizada

```

1 val rdd = spark.sparkContext.parallelize(1 until n)
2 val pointsInsideCircle = rdd.map { i =>
3   val x = random * 2 - 1
4   val y = random * 2 - 1
5   if (x*x + y*y <= 1) 1 else 0
6 }
7 val count = pointsInsideCircle.reduce(_ + _)
8 println(s"Pi is roughly ${4.0 * count / (n - 1)}")

```

Figura 5.3 – Exemplo de aplicação Spark que estima o valor de π .

Fonte: Baseado em (SPARK, 2019)

nos experimentos da taxonomia de problemas de desempenho. Na linha 1, é feita a leitura do conjunto de dados a partir de um endereço que pode ser um arquivo no HDFS. Na linha 2, é chamada a transformação de mapeamento *flatMap* para dividir os textos de entrada em frases separadas por ponto final, fazendo uso de uma expressão regular para isso. Essa divisão é necessária porque *n-grams*, palavras contíguas em um texto, devem fazer parte de uma mesma frase. Em seguida, é feita novamente a chamada da transformação *flatMap* na linha 3 em que é passada a função *nGrams* que recebe como entrada o número *n* e a *string* de uma frase e retorna uma lista de *n-grams*. Na linha 4, a transformação de filtragem *filter* é chamada para remover *n-grams* vazios ou que possuam um número de palavras diferente de *n*. Na linha 5, o RDD de *n-grams* é transformado em um RDD do tipo chave/valor em que a chave é o próprio *n-gram* e o valor o número inteiro 1. A frequência de cada *n-gram* é calculada na linha 6 em que é chamada a transformação de agregação *reduceByKey* que agrupa os valores pela chave (*n-gram*) e depois soma os valores, o que no caso resulta na quantidade de vezes que cada *n-gram* aparece. A linha 7 finaliza a aplicação com a ação *saveAsTextFile* que inicia a execução das transformações e salva o seu conteúdo.

```

1 val input = sparkContext.textFile(inputURL)
2 val sentences = input.flatMap(x => x.split("(?<=[a-z])\\.\\.\\.s+"))
3 val ngrams = sentences.flatMap(nGrams(n, _))
4 val ngramsFiltered = ngrams.filter(l => l.filter(w => !w.trim.isEmpty).
   size == n && l != List.fill(n - 1)(start) :+ end)
5 val ngramsPairs = ngramsFiltered.map(x => (x, 1))
6 val ngramsCount = ngramsPairs.reduceByKey((x, y) => x + y)
7 ngramsCount.saveAsTextFile(outputURL)

```

Figura 5.4 – Exemplo de aplicação Spark que computa *n-grams* e calcula suas frequências.

A Figura 5.5 apresenta um exemplo com as transformações de conjuntos de Spark baseado no exemplo apresentado em (KARAU; WARREN, 2017). Nele, dois *arrays* de números inteiros (linhas 1 e 2) são paralelizados para criarem RDDs (linhas 3 e 4). Em seguida, são aplicadas as transformações de interseção (linha 5) e subtração (linha 6) entre os dois RDDs. Após, é feita uma união entre os resultados das últimas transformações

(linha 7) e, por último, é feita uma verificação para constatar que o resultado da união é diferente do primeiro RDD (linha 8).

```

1 val a = Array(1, 2, 2, 3, 3, 3, 4, 4, 4, 4)
2 val b = Array(3, 4)
3 val rddA = sc.parallelize(a)
4 val rddB = sc.parallelize(b)
5 val intersection = rddA.intersection(rddB)
6 val subtraction = rddA.subtract(rddB)
7 val union = intersection.union(subtraction)
8 assert(!rddA.collect().sorted.sameElements(union.collect().sorted))

```

Figura 5.5 – Exemplo de aplicação Spark com transformações de conjuntos.

A Figura 5.6 apresenta um exemplo de uma aplicação que contém uma junção e ordenação baseado no *Join Query* do benchmark do *AMPLab Big Data Benchmark* (AMPLAB, 2019) (também utilizada nos experimentos da taxonomia de problemas de desempenho, nas aplicações 1 e 2). Entre as linhas 1 e 4, são lidos os conjuntos de dados e feito mapeamentos para transformar os dados de texto para objetos específicos. Na linha 5, é feita uma filtragem em *userVisits* para remover elementos com base em um período de datas e, em seguida, o RDD é transformado para o tipo chave valor. Na linha 6, o RDD *rankings* é mapeado para o tipo chave/valor. A junção entre os dois RDDs é feita na linha 7. Os dados são mapeados para projetar apenas os campos que serão agregados na linha 8 e, em seguida, são agrupados e agregados na linha 9. Os resultados da agregação são ordenados na linha 10 e o seu conteúdo é salvo na linha 11.

```

1 val rankingsLines = sparkContext.textFile(inputRankingsURL)
2 val rankings = rankingsLines.map(parseRankings)
3 val userVisitsLines = sparkContext.textFile(inputUserVisitsURL)
4 val userVisits = userVisitsLines.map(parseUserVisits)
5 val subqueryUV = userVisits.filter(u => u.visitDate.after(date1) && u.
    visitDate.before(date2)).map(u => (u.destURL, u))
6 val subqueryR = rankings.map(r => (r.pageURL, r))
7 val subqueryJoin = subqueryR.join(subqueryUV)
8 val subquerySelect = subqueryJoin.values.map(v => (v._2.sourceIP, (v._2.
    adRevenue, v._1.pageRank)))
9 val subqueryAggregation = subquerySelect.groupByKey().map(v => (v._1, v._2
    .map(_._1).sum, v._2.map(_._2).sum / v._2.map(_._2).size))
10 val results = subqueryAggregation.sortBy(_._2, false)
11 results.saveAsTextFile(outputURL)

```

Figura 5.6 – Exemplo de aplicação Spark com operações de junção e ordenação.

Fonte: Baseado em (AMPLAB, 2019)

A Figura 5.7 apresenta um exemplo de uma aplicação Spark que contém chamadas às ações numéricas de Spark (ações que operam sob RDDs de tipos numéricos) baseado em um exemplo apresentado em (ZAHARIA et al., 2015). Neste exemplo, são removidos valores atípicos (*outliers*) de um conjunto de dados numéricos com base na média e desvio

padrão deste conjunto. Na linha 1, o RDD *distance* que possui um conjunto de distâncias em formato de *String* é mapeado para um RDD numérico e persistido em memória para ser reutilizado em diferentes ações. Na linha 2, a ação *stdev*, que só é habilitada em RDDs numéricos, é chamada para se obter o desvio padrão do conjunto. A média do conjunto é obtida na linha 3 através da ação *mean*. Os valores atípicos são removidos na linha 4 através de uma transformação de filtragem que remove elementos com base no desvio padrão e média. O conteúdo desse RDD é impresso na linha 5 após a chamada da ação *collect*.

```
1 val distanceDoubles = distance.map(string => string.toDouble).persist()
2 val stddev = distanceDoubles.stdev()
3 val mean = distanceDoubles.mean()
4 val reasonableDistances = distanceDoubles.filter(x => math.abs(x-mean) < 3
    * stddev)
5 println(reasonableDistance.collect().toList)
```

Figura 5.7 – Exemplo de aplicação Spark com ações numéricas.

Fonte: Baseado em (ZAHARIA et al., 2015)

A Figura 5.8 apresenta um exemplo de uma aplicação Spark que utiliza acumuladores baseado em um exemplo apresentado em (GANELIN et al., 2016). Neste exemplo, variáveis acumuladoras são utilizadas para contar números pares e ímpares em um RDD. Na linha 1, é criado um RDD a partir da paralelização de um *array* de números. Acumuladores são criados nas linhas 2 e 3. Entre as linhas 4 e 7, é chamada a ação *foreach* que aplica uma função para cada elemento do RDD. Nesta função, os acumuladores são incrementados para contar números pares e ímpares. Os valores dos acumuladores são impressos nas linhas 8 e 9.

```
1 val myRdd = sc.parallelize(Array(1,2,3,4,5,6,7))
2 val evenNumbersCount = sc.longAccumulator(0)
3 val unevenNumbersCount = sc.longAccumulator(0)
4 myRdd.foreach(element => {
5   if (element % 2 == 0) evenNumbersCount += 1
6   else unevenNumbersCount +=1
7 })
8 println(s" Even numbers ${ evenNumbersCount.value }")
9 println(s" Uneven numbers ${ unevenNumbersCount.value }")
```

Figura 5.8 – Exemplo de aplicação Spark que utiliza um acumulador.

Fonte: Baseado em (GANELIN et al., 2016)

A Figura 5.9 apresenta um exemplo de implementação de um acumulador personalizado de Spark baseado no exemplo apresentado em (GANELIN et al., 2016). Neste exemplo, foi criado um acumulador para acumular nomes de arquivos que foram processados com erros. Para se implementar um acumulador, deve-se estender o tipo *AccumulatorV2* (linha 1) e prover implementações para operações padrão. Entre as principais

operações que compõem o comportamento do acumulador, estão a *add* (linhas 14 a 18), que adiciona um novo elemento no acumulador, e a *merge* (linha 25 a 27) que une o conteúdo de dois acumuladores.

```

1  class ErrorFilesAccum extends AccumulatorV2[String, String] {
2    var initialValue: String = ""
3
4    def value = initialValue
5
6    def reset(): Unit = {
7      initialValue = ""
8    }
9
10   def isZero: Boolean = {
11     initialValue.isEmpty
12   }
13
14   def add(s: String): Unit = {
15     var str = ""
16     if(!isZero) str = ", " + s else s
17     initialValue = initialValue + s
18   }
19
20   def copy(): ErrorFilesAccum = {
21     val newAcc = new ErrorFilesAccum()
22     newAcc.initialValue = value
23   }
24
25   def merge(other: AccumulatorV2[String, String]) = {
26     add(other.value)
27   }
28 }

```

Figura 5.9 – Exemplo de implementação de um acumulador personalizado.

Fonte: Baseado em (GANELIN et al., 2016)

A Tabela 5.1 apresenta exemplos de defeitos que podem aparecer nos exemplos de aplicações apresentados. Nela, é possível ver um exemplo para cada defeito apresentado na taxonomia de defeitos funcionais.

5.3 Discussões

Neste trabalho, utilizamos uma metodologia iterativa e incremental para a análise de diferentes fontes para identificar e classificar defeitos em aplicações Spark. Como resultado, desenvolvemos uma taxonomia de defeitos funcionais que descreve alguns dos possíveis defeitos que podem ser encontrados nas principais operações de transformação e ação de Spark, assim como no uso e desenvolvimento das variáveis compartilháveis acumuladores. Os defeitos descritos nesta taxonomia foram mostrados através de exemplos de aplicações e possíveis defeitos sintetizados na Tabela 5.1. Esperamos que os defeitos

Tabela 5.1 – Exemplos de defeitos e suas classificações segundo a Taxonomia de Defeitos Funcionais de Spark.

Fig.	Linhas	Defeito	Classificação	Considerações
5.2	2	<code>filter(!_startsWith("ERROR"))</code>	F1.4.1	A definição do predicado para filtrar logs de erros está incorreta.
	3	<code>map(_split('\n')(2))</code>	F1.3.1	A função de transformação está separando a string com um separador incorreto.
	2 a 3	Inverter a ordem das duas transformações.	F1.1	Ao inverter a ordem das duas transformações o resultado final é diferente.
5.3	7	<code>pointsInsideCircle.reduce(_ - _)</code>	F2.2.1 e F2.2.2	Além da definição da agregação estar incorreta, ela também não é comutativa e associativa.
5.4	5	<code>ngramsFiltered.map(x =>(x, 0))</code>	F1.3.1.1	O elemento está sendo mapeado para uma chave/valor com o valor errado.
	6	<code>ngramsPairs.reduceByKey((x, y) =>x - y)</code>	F1.5.1 e F1.5.2	A definição da transformação de agregação está incorreta e, além disso, ela é não comutativa e não associativa.
5.5	5	<code>rddA.union(rddB)</code>	F1.2.2	Foi chamada a transformação union ao invés de intersection.
	8	<code>rddA.distinct().collect().sorted == union.collect().sorted</code>	F1.2.1	Esse predicado é falso porque a operação union possui elementos repetidos.
5.6	7	<code>subqueryR.rightOuterJoin(subqueryUV)</code>	F1.7.1	Foi aplicado o tipo incorreto de junção.
	10	<code>subqueryAggregation.sortBy(_._2, true)</code>	F1.6.1	A ordenação está sendo feita de forma ascendente enquanto que no original é descendente.
5.7	2	<code>distanceDoubles.max()</code>	F2.1	A ação max foi chamada no lugar de stdev.
5.8	2	<code>var evenNumbersCount = 0</code>	F3.1	A variável foi declarada como uma variável mutável normal e não um acumulador.
5.9	14 a 18	Implementação de add com exclusão da declaração de if.	F3.2.1 e F3.2.2	A própria implementação do acumulador não é comutativa uma vez que ao inverter a ordem dos elementos a serem adicionados a string resultante seria diferente. Além disso, a remoção da cláusula if faz o resultado ser incorreto.

descritos aqui possam ser utilizados como referência para o desenvolvimento de casos de teste e técnicas de teste para aplicações Spark. Uma das suas possíveis aplicações é no desenvolvimento de *operadores de mutação* que simulam defeitos para permitir a aplicação de *Teste de Mutação* em Spark.

Os resultados deste trabalho foram obtidos de uma maneira empírica através do estudo de fontes na literatura, documentação de Spark, desenvolvimento de aplicações e análise manual de códigos fontes de aplicações Spark. É importante ressaltar que esse estudo foi feito de maneira limitada dada a dificuldade em encontrar repositórios públicos de projetos que utilizam Spark. Esses projetos poderiam ser fontes mais completas de estudos uma vez que poderiam reportar defeitos que apareceram na prática por outras pessoas além do autor. Além disso, questões subjetivas da análise de possíveis defeitos e suas classificações por parte do autor se mostram como possíveis ameaças para a validação deste trabalho. Por esse motivo, se faz necessário mais estudos em diferentes fontes sobre defeitos em aplicações Spark para validar e melhorar a taxonomia desenvolvida. Apesar disso, é importante destacar que não foram encontrados em nossas pesquisas outros trabalhos que buscaram explorar defeitos funcionais em aplicações Spark. Por conta disso, nosso trabalho pode ser considerado pioneiro nesse sentido e utilizado como referência em outros estudos exploratórios sobre defeitos e problemas relacionados a Spark.

Um estudo mais abrangente sobre defeitos e problemas em Spark está sendo desenvolvido atualmente por alunos de mestrado do Programa PPgSC/UFRN. Os mestrandos Denis Albuquerque e Roberta Cynthia, em conjunto com os professores Dr. Umberto de Souza Costa e Dr. Martin Alejandro Musicante, estão fazendo uma análise do conteúdo relacionado a Apache Spark no *Stack Overflow*. Este é uma plataforma de perguntas e respostas na *web* que permite que pessoas postem dúvidas, dificuldades e problemas e recebam respostas e *feedbacks* de outras pessoas, como desenvolvedores e pesquisadores. O *Stack Overflow* é a plataforma recomendada no site oficial de Spark para discussão de problemas e dúvidas no desenvolvimento de aplicações Spark. Por esse motivo, as perguntas e respostas que são relacionadas com Spark postadas no *Stack Overflow* podem ser uma grande fonte de estudos para identificar defeitos em aplicações Spark. Ao todo, existem mais de 49 mil perguntas relacionadas com Spark no *Stack Overflow*, o que impossibilita a análise manual deste. Por esse motivo, estão sendo aplicadas técnicas que visam analisar as perguntas e respostas de forma automática e extrair informações referente às dúvidas e problemas mais recorrentes sobre Spark. Para isso, está sendo aplicado o algoritmo *Latent Dirichlet Allocation* (LDA) (BLEI; NG; JORDAN, 2003) de modelagem probabilística de tópicos no corpus de perguntas e respostas do *Stack Overflow* relacionadas a Spark. O objetivo deste estudo é estabelecer uma hierarquia de assuntos relacionados a Spark no *Stack Overflow* e utilizar essa hierarquia para filtrar perguntas específicas em futuras análises. Com os resultados desse estudo, pretendemos fazer uma análise mais precisa sobre defeitos e problemas recorrentes em Spark de modo a comparar com os resultados obtidos

no estudo apresentado neste capítulo.

6 Um Modelo Geral para Programas de Processamento de Big Data

Este capítulo apresenta um modelo de generalização para programas de processamento de Big Data. Este modelo foi criado a partir de características em comum identificadas nos sistemas Apache Spark, Hadoop MapReduce, Dryad e DryadLINQ, Nephele/PACTs e o FlumeJava e Apache Beam, que foram os sistemas apresentados no Capítulo 3. Esses sistemas possuem semelhanças na forma em que um programa é representado e as operações sobre dados que podem ser feitas. A partir dessas semelhanças, definimos um modelo que procura abranger as principais características funcionais de programas de processamento de Big Data. Na maioria dos sistemas estudados, um programa é representado a partir de um grafo acíclico orientado (DAG) em que os seus vértices representam operações sobre conjuntos de dados. Essas operações, por sua vez, definem manipulações que transformam um conjunto de dados de entrada em um conjunto de dados de saída. Nosso modelo é definido a partir de dois formalismos, um formalismo baseado em *Redes de Petri* (MURATA, 1989) para definir a DAG que representa um programa, que vamos denominar como sendo o seu *fluxo de dados*, e em *Álgebra de Monoides* (FEGARAS, 2017) para definir as operações existentes no modelo, que vamos denominar como *transformações*.

Este capítulo está organizado da seguinte forma: a Seção 6.1 apresenta a formalização para o fluxo de dados; a formalização para as transformações e conjuntos de dados é apresentada na Seção 6.2; por último, a Seção 6.3 apresenta um exemplo de aplicação do modelo.

6.1 Fluxo de Dados

Para definir o DAG que representa o fluxo de dados de programas de processamento de Big Data, nos baseamos no modelo formal de grafos de fluxo de dados apresentado em (KAVI; BUCKLES; BHAT, 1986). Esse modelo, por sua vez, é baseado no formalismo de *Redes de Petri* (MURATA, 1989). Nele, um grafo de fluxo de dados é representado como um grafo direcionado bipartido que possui dois tipos de vértices, as ligações (*links*) e os atores (*actors*). Atores representam operações sobre dados, enquanto ligações representam locais reservados que recebem dados de um ator e transmitem dados para um ou mais atores. Atores e ligações são ligados através de arestas que representam canais de comunicação entre os vértices.

No nosso modelo, um programa de processamento de Big Data P é definido como

um grafo de fluxo de dados que possui dois tipos de vértices, os conjuntos de dados (D), que representam as ligações, e as transformações (T), que representam os atores, que são ligados por arestas (E):

$$P = \langle D \cup T, E \rangle$$

Arestas são definidas como um subconjunto do produto cartesiano de conjunto de dados para transformações, que caracterizam o conjunto de dados de entrada de uma transformação, ou do produto cartesiano de transformações para conjuntos de dados, que caracterizam o conjunto de dados de saída de uma transformação:

$$E \subseteq (D \times T) \cup (T \times D)$$

Um elemento de D representa um conjunto de dados distribuídos, enquanto que um elemento de T representa uma operação de transformação que recebe um ou mais conjuntos de dados como entrada e produz um conjunto de dados como saída. Além disso, os conjuntos D e T são disjuntos, o que significa que não existem elementos que pertencem a ambos ao mesmo tempo, e finitos, cada conjunto possui uma quantidade finita de elementos:

$$D = \{d_1, d_2, \dots, d_n\}$$

$$T = \{t_1, t_2, \dots, t_m\}$$

Consideramos dois subconjuntos de D , um que representa os conjuntos de dados de entrada (R) de um programa, que são criados a partir de uma fonte de dados externa, como a partir da leitura de dados em um sistema de arquivos distribuídos tipo o *HDFS*, e um que representa os conjuntos de dados de saída (C) de um programa, que representam conjuntos de dados que serão salvos ou coletados em algum coletor de dados externo, como ser salvo no *HDFS*. Por questão de simplicidade, não definimos no nosso modelo operações que geram os conjuntos de dados em R ou que coletam os conjuntos de dados em C , assumimos que os vértices nesses conjuntos representam as entradas e saídas, respectivamente, do nosso programa P . R e C são definidos como segue:

$$R = \{d \in D \mid \forall t \in T. (t, d) \notin E\}$$

$$C = \{d \in D \mid \forall t \in T. (d, t) \notin E\}$$

O conjunto de conjuntos de dados de entrada de uma transformação t , e o conjunto de conjuntos de dados de saída de uma transformação t são denotados $I(t)$ e $O(t)$:

$$I(t) = \{d \in D \mid (d, t) \in E\}$$

$$O(t) = \{d \in D \mid (t, d) \in E\}$$

De forma semelhante, o conjunto de transformações que deram origem a um conjunto de dados d e o conjunto de transformações que recebem o conjunto de dados d como entrada são denotados $I(d)$ e $O(d)$:

$$I(d) = \{t \in T \mid (t, d) \in E\}$$

$$O(d) = \{t \in T \mid (d, t) \in E\}$$

Com essas definições, as transformações e conjuntos de dados no programa devem satisfazer às seguintes condições:

$$\begin{array}{ll} 1 \leq |I(t)| \leq 2 & \text{para todo } t \in T \\ |O(t)| = 1 & \text{para todo } t \in T \\ 0 \leq |I(d)| \leq 1 & \text{para todo } d \in D \\ |O(d)| \geq 0 & \text{para todo } d \in D \end{array}$$

De forma simplificada, essas condições representam que uma transformação deve receber um ou dois conjuntos de dados como entrada ($|I(t)| = 1$ ou $|I(t)| = 2$) e produzir exatamente um conjunto de dados como saída ($|O(t)| = 1$), e que um conjunto de dados deve pertencer a R ($|I(d)| = 0$) ou ter sido gerado por uma transformação ($|I(d)| = 1$) e deve pertencer a C ($|O(d)| = 0$) ou ser entrada para uma ou mais transformações ($|O(d)| > 0$). Vamos denominar de *transformações unárias* as transformações que operam sobre um único conjunto de dados ($|I(t)| = 1$) e de *transformações binárias* as transformações que operam sobre dois conjuntos de dados ($|I(t)| = 2$).

A semântica dos conjuntos de dados em D e transformações em T é apresentada na seção seguinte. Consideramos que transformações binárias recebem os conjuntos de dados de entrada de maneira ordenada. Assim sendo, vamos considerar as funções $input_type_1(t)$, do tipo $T \rightarrow \tau_1$, $input_type_2(t)$, do tipo $T \rightarrow \tau_2$, e $output_type(t)$, do tipo $T \rightarrow \tau_3$, para denotar respectivamente o tipo τ_1 do primeiro conjunto de dados de entrada de uma transformação t (para o caso de transformações unárias e binárias), o tipo τ_2 do segundo conjunto de dados de entrada de uma transformação t (apenas no caso em que t é uma transformação binária) e o tipo τ_3 do conjunto de dados de saída de uma transformação t . Além disso, vamos considerar a função $type(d)$, do tipo $D \rightarrow \tau$, que denota o tipo (τ) de um conjunto de dados. A partir dessas funções, também definimos que as transformações e conjuntos de dados em um programa devem satisfazer às seguintes condições:

$$\begin{aligned} \forall t \in T, d \in D. (d, t) \in E &\Rightarrow type(d) = input_type_1(t) \vee type(d) = input_type_2(t) \\ \forall t \in T, d \in D. (t, d) \in E &\Rightarrow output_type(t) = type(d) \end{aligned}$$

Por fim, com base nos tipos de operações existentes nos sistemas de processamento de Big Data discutidos, observamos a existência de dois tipos de transformações, aquelas que requerem uma reorganização (*shuffle*) dos dados no cluster e aquelas que não. No nosso modelo, vamos chamar o primeiro grupo de transformações *shuffle*, representado pelo conjunto S , e o segundo grupo de transformações lineares, representado pelo conjunto L :

$$S \subseteq T$$

$$L \subseteq T$$

As transformações em S e L seguem os conceitos de transformações amplas e estreitas definidas no Apache Spark (ZAHARIA et al., 2012). A semântica de uma transformação *shuffle* e linear é discutida na seção seguinte. A Tabela 6.1 apresenta um resumo das definições apresentadas acima.

Tabela 6.1 – Descrição das representações do fluxo de dados de um programa de processamento de *Big Data*.

Representação	Descrição
P	Representação de um programa de processamento de <i>Big Data</i> que possui D , T e E .
D	Conjunto de conjuntos de dados de um programa P .
T	Conjunto de operações de transformações sobre conjuntos de dados de um programa P .
E	Conjunto de arestas que ligam os conjuntos de dados em D e as transformações em T de um programa P .
R	Subconjunto de D que representa os conjuntos de dados de entrada de um programa P .
C	Subconjunto de D que representa os conjuntos de dados de saída de um programa P .
S	Subconjunto de T que representa transformações que requerem redistribuição de dados (<i>shuffle</i>).
L	Subconjunto de T que representa transformações que não requerem redistribuição de dados.
$I(t)$	Conjunto de conjunto de dados de entrada de uma transformação t .
$O(t)$	Conjunto de conjunto de dados de saída de uma transformação t .
$I(d)$	Conjunto de transformações que geram o conjunto de dados d .
$O(d)$	Conjunto de transformações que recebem o conjunto de dados d como entrada.

A partir das definições apresentadas, conseguimos representar os conceitos de DAG que são utilizados nos sistemas Apache Spark, FlumeJava/Apache Beam, DryadLINQ e PACTs.

6.2 Conjunto de Dados e Transformações

Para definir a semântica dos conjuntos de dados em D e os tipos de transformações em T , fazemos uso da *Álgebra de Monoides* (FEGARAS, 2017; FEGARAS, 2019). Inicialmente, faremos uma breve introdução à Álgebra de Monoides e em seguida apresentamos os conjuntos de dados e transformações em termos dela.

6.2.1 Álgebra de Monoides

A Álgebra de Monoides foi proposta em (FEGARAS, 2017; FEGARAS, 2019) como um formalismo algébrico para operações de computação distribuída centrada em dados baseados em monoides e homomorfismo de monoides. Um monoide é uma estrutura algébrica que possui um conjunto S , uma operação associativa $\oplus$ do tipo $S \times S \rightarrow S$ e um elemento neutro $e \in S$. A estrutura $(S, \oplus, e)$ é considerada um monoide se e somente se:

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z \quad \text{para todo } x, y, z \in S \text{ (}\oplus \text{ é associativo)}$$

$$x \oplus e = x = e \oplus x \quad \text{para todo } x \in S \text{ (}e \text{ é um elemento neutro)}$$

Um monoide pode ser identificado através de sua operação binária $\oplus$, sendo referenciado apenas por $\oplus$. Além disso, vamos denotar $1_{\oplus}$ como sendo o elemento neutro de $\oplus$ e $\mathbb{T}_{\oplus}$ como o tipo do conjunto S de $\oplus$. Um homomorfismo de monoides é uma função H entre dois monoides $\otimes$ e $\oplus$ que respeita as seguintes propriedades:

$$H(X \otimes Y) = H(X) \oplus H(Y) \quad \text{para todos } X \text{ e } Y \text{ do tipo } \mathbb{T}_{\otimes}$$

$$H(1_{\otimes}) = 1_{\oplus}$$

A Álgebra de Monoides utiliza o conceito de homomorfismo de monoides para definir operações sobre coleções monoides. Uma coleção monoide é uma coleção do tipo $T(\alpha)$, com dados de um certo tipo α , que possui uma operação associativa $\otimes$, um elemento neutro $1_{\otimes}$, e uma função injetora unitária $\mathbb{U}_{\otimes}(x)$ do tipo $\alpha \rightarrow T(\alpha)$. Um tipo de coleção monoide é o pacote (*bag*), que é uma coleção de dados de um tipo α (vamos denotar $Bag[\alpha]$ como uma bag de elementos do tipo α), que possui a operação associativa $\uplus$, que faz a união entre duas bags, o elemento neutro $\{\{\}$, que representa uma bag vazia, e a função injetora unitária $\mathbb{U}_{\otimes}(x) = \{\{x\}\}$. Por exemplo, $\{\{1\}\} \uplus \{\{2\}\} \uplus \{\{3\}\} \uplus \{\{\}$ resulta na bag $\{\{1, 2, 3\}\}$. Além de ser associativa, uma coleção bag também é comutativa, de forma que não importa a ordem dos elementos em uma união, o seu resultado é sempre o mesmo, e a ordem dos elementos em uma bag não a diferencia de outra ($\{\{1, 2, 3\}\} = \{\{3, 2, 1\}\}$), além de suportar elementos repetidos. Outro tipo de coleção monoide é a lista, que pode ser considerada uma bag ordenada (não associativa), que é uma coleção de um tipo α (vamos denotar $List[\alpha]$), que possui a operação associativa $++$, o elemento neutro $[]$ e a função injetora $\mathbb{U}_{\otimes}(x) = [x]$. Por exemplo, $[1] ++ [2] ++ [3] ++ [] = [1, 2, 3]$ e $[1, 2, 3] \neq [3, 2, 1]$. As

características das coleções monoides permitem que elas representem conjuntos de dados distribuídos, assim como conjuntos de dados locais.

As operações na Álgebra de Monoides são definidas como homomorfismos de monoides sobre coleções de dados monoides. Essas operações definem os tipos de manipulações que podem ser feitas nessas coleções. Vamos descrever algumas das principais operações algébricas da Álgebra de Monoides que representam operações sobre coleções de dados. Operações específicas sobre os elementos das coleções, que representam as funções definidas pelo desenvolvedor, são capturadas como expressões funcionais passadas como argumento para algumas das operações (CHLYAH et al., 2019).

A operação **flatMap** recebe uma função f do tipo $\alpha \rightarrow Bag[\beta]$ e uma coleção X do tipo $Bag[\alpha]$ como entrada e resulta em uma coleção do tipo $Bag[\beta]$ que é resultado da união de todos os resultados de f aplicado aos elementos de X . Essa operação captura a essência do processamento paralelo de dados uma vez que f pode ser executada de forma paralela em diferentes partições de dados de uma coleção distribuída. Por exemplo, considere a função f que recebe um número inteiro x e retorna uma coleção contendo seu número antecessor e seu número sucessor ($f(x) = \{\{x-1, x+1\}\}$) e a coleção $X = \{\{1, 2, 3\}\}$. Então, $\text{flatMap}(f, X) = f(1) \uplus f(2) \uplus f(3) = \{\{0, 2\}\} \uplus \{\{1, 3\}\} \uplus \{\{2, 4\}\} = \{\{0, 2, 1, 3, 2, 4\}\}$.

As operações **groupby** e **cogroup** capturam o processo de redistribuição (*shuffling*) de dados uma vez que representam uma reorganização e agrupamento dos dados, podendo implicar na movimentação de dados em coleções distribuídas. A operação **groupby** agrupa os elementos de uma coleção $Bag[\kappa \times \alpha]$ através do primeiro componente (chave) do tipo κ e resulta em uma coleção $Bag[\kappa \times Bag[\alpha]]$ em que o segundo componente é uma coleção contendo todos os elementos do tipo α que eram associados a uma mesma chave na coleção inicial. A operação **cogroup** trabalha de forma semelhante à **groupby**, mas opera sobre duas coleções, uma do tipo $Bag[\kappa \times \alpha]$ e outra do tipo $Bag[\kappa \times \beta]$. O seu resultado é uma coleção do tipo $Bag[\kappa \times (Bag[\alpha] \times Bag[\beta])]$ em que o segundo componente são duas coleções, uma do tipo α e outra do tipo β , contendo os elementos que eram associados a uma mesma chave nas coleções de entrada. Por exemplo, considere as coleções $X = \{\{(k_1, a), (k_2, b), (k_1, c)\}\}$ e $Y = \{\{(k_1, 1), (k_2, 2), (k_3, 3), (k_1, 4)\}\}$. Então, $\text{groupby}(X) = \{\{(k_1, \{\{a, c\}\}), (k_2, \{\{b\}\})\}\}$, $\text{groupby}(Y) = \{\{(k_1, \{\{1, 4\}\}), (k_2, \{\{2\}\}), (k_3, \{\{3\}\})\}\}$ e $\text{cogroup}(X, Y) = \{\{(k_1, (\{\{a, c\}\}, \{\{1, 4\}\})), (k_2, (\{\{b\}\}, \{\{2\}\})), (k_3, (\{\{\}, \{\{3\}\}))\}\}$.

A operação **reduce** representa a agregação dos elementos de uma coleção $Bag[\alpha]$ em um único elemento do tipo α a partir da aplicação de uma função associativa f do tipo $\alpha \rightarrow \alpha \rightarrow \alpha$. Por exemplo, considere a função f que recebe dois números inteiros x e y e retorna a soma dos dois números ($f(x, y) = x + y$) e a coleção $X = \{\{1, 2, 3, 4, 5\}\}$. Então, $\text{reduce}(f, X) = 15$.

A operação **orderby** representa a transformação de uma bag $Bag[\kappa \times \alpha]$ em uma lista $List[\kappa \times \alpha]$ ordenada pela chave do tipo κ que suporta a ordem total $\leq$. Por exem-

plo, considere a coleção $X = \{(1, a), (3, b), (5, c), (2, d), (4, e)\}$. Então, $\text{orderBy}(X) = [(1, a), (2, d), (3, b), (4, e), (5, c)]$.

A álgebra também suporta expressões funcionais, através da expressão lambda $\lambda var.e$ que representa uma função que recebe uma lista de variáveis var e resulta na expressão e (por exemplo, $\lambda x.\{(x, x)\}$, é uma função que recebe x e retorna uma bag contendo a tupla (x, x)). Além de suportar expressões condicionais, através do **if** e_1 **then** e_2 **else** e_3 . As operações e expressões da Álgebra de Monoides podem ser aninhadas e compostas para a definição de operações mais complexas. Mais detalhes sobre a Álgebra de Monoides, como a definição formal de suas operações, semântica e sintaxe, podem ser encontradas em (FEGARAS, 2017), (FEGARAS, 2019) e (CHLYAH et al., 2019).

6.2.2 Conjuntos de Dados

No nosso modelo, consideramos que um conjunto de dados em D pode ser representado por ambos, uma coleção de dados do tipo *bag* ($Bag[\alpha]$) ou uma coleção de dados do tipo lista ($List[\alpha]$). Ambos podem representar coleções de dados distribuídos (FEGARAS, 2019), capturando a essência dos conceitos de *RDD* no Apache Spark, *PCollection* no Apache Beam e *DryadTable* no DryadLINQ. Por questão de simplicidade, vamos definir as transformações do nosso modelo em termos de *bags*, considerando listas apenas nas transformações de ordenação, que são as únicas em que a ordem dos elementos no conjunto de dados é relevante. Além disso, não faremos diferença entre coleções distribuídas ou não para definir nossas transformações uma vez que operações sobre coleções locais são definidas de maneira análoga em coleções distribuídas, assim como a definição de operações sobre *bags* e listas (FEGARAS, 2019).

6.2.3 Transformações

As transformações representam operações sobre conjuntos de dados, de modo que uma transformação pode receber um ou dois conjuntos de dados como entrada e produz um conjunto de dados como saída. Transformações também podem receber outros tipos de parâmetros, como funções, que representam as operações sobre dados definidas pelo desenvolvedor, assim como valores aritméticos ou lógicos. Uma transformação t no conjunto de transformações T de um programa de processamento de Big Data P é caracterizada pelas operações descritas a seguir, os tipos dos seus conjuntos de dados de entrada, o tipo do conjunto de dados de saída e os seus parâmetros.

Uma transformação t está em S (transformação *shuffle*) se esta for definida em termos das operações **groupby**, **cogroup**, **reduce** ou **orderBy** da Álgebra de Monoides, uma vez que essas operações exigem uma redistribuição dos dados (FEGARAS, 2017). Uma transformação t está em L (transformação linear) se esta for definida apenas em

termos de **flatmap**. A seguir definimos as transformações do nosso modelo em termos das operações da Álgebra de Monoides. Separamos as transformações em categorias seguindo os tipos de operações que foram observados nos sistemas de processamento de Big Data estudados.

Mapeamento: transformações desse tipo representam operações que mapeiam os valores de um conjunto de dados de entrada para os valores de um conjunto de dados de saída a partir da aplicação de uma função de transformação. Definimos duas transformações de mapeamento, a *flatMap* e a *map*.

A transformação *flatMap* é sinônima à operação **flatmap** da Álgebra de Monoides e possui o mesmo comportamento da operação *Map* no modelo MapReduce, em que dada a função f , que pode ser definida pelo desenvolvedor, mapeia um elemento do tipo α para uma coleção com zero ou mais elementos do tipo β :

$$\begin{aligned} flatMap &:: (\alpha \rightarrow Bag[\beta]) \rightarrow Bag[\alpha] \rightarrow Bag[\beta] \\ flatMap(f, D) &= \mathbf{flatmap}(f, D) \end{aligned}$$

A operação *map* é semelhante à *flatMap*, mas a sua função de transformação f faz um mapeamento de um elemento do tipo α para um elemento do tipo β , ao invés de resultar em uma coleção do tipo $Bag[\beta]$, como no caso de *flatMap*. Uma vez que f não resulta em uma coleção, é necessário transformar o seu resultado em uma coleção do tipo $Bag[\beta]$ para podermos aplicar a operação **flatmap**. Para isso, criamos uma expressão lambda que recebe um elemento x do conjunto de dados de entrada e que tem como resultado uma coleção do tipo $Bag[\beta]$ contendo apenas o resultado da aplicação de f em x ($\lambda x. \{f(x)\}$). Dessa forma, *map* é definida como:

$$\begin{aligned} map &:: (\alpha \rightarrow \beta) \rightarrow Bag[\alpha] \rightarrow Bag[\beta] \\ map(f, D) &= \mathbf{flatmap}(\lambda x. \{f(x)\}, D) \end{aligned}$$

Por exemplo, considere a função f que retorna o sucessor de um número inteiro x ($f(x) = x + 1$) e a coleção $D = \{1, 2, 3\}$. Então, $map(f, D) = \{2, 3, 4\}$.

Filtragem: uma transformação de filtragem mapeia os elementos de um conjunto de dados de entrada para um conjunto de dados de saída a partir de uma função de predicado que determina se o elemento deve entrar ou não no conjunto de saída. No nosso modelo, definimos a transformação *filter* que recebe a função p que mapeia um elemento do conjunto de dados para um valor lógico (*boolean*), representando se este deve pertencer ou não ao conjunto de saída. Para definir *filter*, precisamos verificar se um elemento x

do conjunto de entrada satisfaz a condição em p ($p(x) = \text{true}$). Essa condição é verificada em uma expressão lambda que utiliza uma expressão condicional (**if-then-else**) que resulta na coleção $\{\{x\}\}$, quando $p(x) = \text{true}$, ou em uma coleção vazia ($\{\{\}\}$), quando $p(x) = \text{false}$. Essa expressão lambda é, então, aplicada no conjunto de dados de entrada através da operação **flatMap**:

$$\begin{aligned} \text{filter} &:: (\alpha \rightarrow \text{boolean}) \rightarrow \text{Bag}[\alpha] \rightarrow \text{Bag}[\alpha] \\ \text{filter}(p, D) &= \text{flatMap}(\lambda x. \text{if } p(x) \text{ then } \{\{x\}\} \text{ else } \{\{\}\}, D) \end{aligned}$$

Por exemplo, considere a função de predicado p que verifica se um número inteiro x é maior ou igual a 3 ($p(x) = x \geq 3$) e uma coleção $D = \{\{1, 2, 3, 4, 5\}\}$. Então, $\text{filter}(p, D) = \{\{3, 4, 5\}\}$.

Agrupamento: transformações desse tipo agrupam os elementos de um conjunto de dados de entrada em um conjunto de dados de saída a partir de uma chave. Definimos duas transformações de agrupamento no nosso modelo, a transformação *groupBy*, que aplica uma função k que mapeia elementos da coleção para uma chave do tipo κ e agrupa os elementos do conjunto a partir das chaves geradas, e a transformação *groupByKey*, que opera sobre um conjunto de dados do tipo chave/valor $\text{Bag}[\kappa \times \alpha]$ e agrupa os elementos pela chave do tipo κ .

Ambas as operações são definidas em termos da operação **groupby** da Álgebra de Monoides, que agrupa os elementos de um conjunto do tipo chave/valor $\text{Bag}[\kappa \times \alpha]$ a partir da chave do tipo κ e que resulta em uma coleção do tipo chave/valor $\text{Bag}[\kappa \times \text{Bag}[\alpha]]$ em que o valor é a coleção de elementos do tipo α associados com uma chave do tipo κ . A transformação *groupBy* recebe uma função k que gera uma chave do tipo κ a partir de um elemento do tipo α e uma coleção D do tipo $\text{Bag}[\alpha]$ como entrada. Para agrupar os elementos de D de acordo com as chaves geradas por k , é necessário primeiro gerar elementos do tipo chave/valor em que a chave é a aplicação de k sobre o valor que é um elemento de D . Isso é feito a partir da aplicação da operação **flatMap** passando como função a expressão lambda que recebe x e gera uma coleção contendo o elemento chave/valor $(k(x), x)$ ($\lambda x. \{\{(k(x), x)\}\}$) e a coleção D . O resultado dessa aplicação de **flatMap** é, então, aplicado na operação **groupby** que agrupa os elementos pela chave. Já a operação *groupByKey*, possui o mesmo comportamento da operação **groupby**. Dessa forma, *groupBy* e *groupByKey* são definidos como:

$$\begin{aligned}
groupBy &:: (\alpha \rightarrow \kappa) \rightarrow Bag[\alpha] \rightarrow Bag[\kappa \times Bag[\alpha]] \\
groupBy(k, D) &= \mathbf{groupby}(\mathbf{flatmap}(\lambda x. \{(k(x), x)\}, D)) \\
groupByKey &:: Bag[\kappa \times \alpha] \rightarrow Bag[\kappa \times Bag[\alpha]] \\
groupByKey(D) &= \mathbf{groupby}(D)
\end{aligned}$$

Para exemplificar, vamos considerar a função k que recebe um número inteiro x e retorna o próprio número x ($k(x) = x$), o conjunto de dados $D_1 = \{1, 2, 3, 2, 3, 3\}$, e o conjunto de dados $D_2 = \{(1, a), (2, b), (3, c), (1, e), (2, f)\}$. Então, a aplicação de $groupBy$ e $groupByKey$ nesses conjuntos resulta em:

$$\begin{aligned}
groupBy(k, D_1) &= \{(1, \{1\}), (2, \{2, 2\}), (3, \{3, 3, 3\})\} \\
groupByKey(D_2) &= \{(1, \{a, e\}), (2, \{b, f\}), (3, \{c\})\}
\end{aligned}$$

Conjuntos: essas transformações representam operações inspiradas em operações de conjuntos matemáticos. Essas transformações operam sobre dois conjuntos de dados de um mesmo tipo e resultam em um novo conjunto de dados que pode conter elementos dos dois conjuntos, no caso das operações que representam a união e intersecção entre conjuntos, ou de um único conjunto, no caso da operação que representa a diferença entre conjuntos. A definição dessas transformações foram baseadas nas definições apresentadas em (FEGARAS, 2019).

A transformação *union* representa a união dos elementos de dois conjuntos de dados em um único conjunto. Essa operação é representada de forma simples através do operador de união de *bags* ($\uplus$):

$$\begin{aligned}
union &:: Bag[\alpha] \rightarrow Bag[\alpha] \rightarrow Bag[\alpha] \\
union(D_x, D_y) &= D_x \uplus D_y
\end{aligned}$$

Por exemplo, considere a coleção $D_1 = \{1, 2, 3\}$ e a coleção $D_2 = \{2, 3, 4, 5\}$. Então, $union(D_1, D_2) = \{1, 2, 3, 2, 3, 4, 5\}$

Uma vez que essa operação de união permite elementos repetidos, dada as características da coleção *bag*, diferente da operação de união entre conjuntos matemáticos, definimos a operação *distinct* para permitir que elementos duplicados sejam removidos da coleção. Para definir *distinct*, devemos agrupar os elementos do conjunto tendo o próprio elemento como chave e depois pegar apenas a chave, dessa forma, os valores que estão no grupo, que é a própria chave repetida, são descartados, ficando assim apenas os elementos

sem repetição. Para isso, primeiro devemos transformar os elementos do conjunto em elementos do tipo chave/valor em que a chave e o valor são o próprio elemento. Fazemos isso através da aplicação de uma operação **flatmap** ($t_1(D) = \text{flatmap}(\lambda x. \{(x, x)\}, D)$). Em seguida, agrupamos o resultado desse conjunto com a operação **groupby**, que gera um conjunto chave/valor em que a chave é o próprio elemento e o valor é uma coleção contendo os elementos repetidos ($t_2(D) = \text{groupby}(t_1(D))$). Por fim, o resultado desse agrupamento é aplicado em um novo mapeamento com **flatmap** que aplica uma função que pega o elemento chave/valor e retorna apenas a chave ($\text{flatmap}(\lambda(k, g). \{k\}, t_2(D))$). Dessa forma, *distinct* é definido como segue:

$$\begin{aligned} \text{distinct} &:: \text{Bag}[\alpha] \rightarrow \text{Bag}[\alpha] \\ \text{distinct}(D) &= \text{flatmap}(\lambda(k, g). \{k\}, t_2(D)) \\ t_1(D) &= \text{flatmap}(\lambda x. \{(x, x)\}, D) \\ t_2(D) &= \text{groupby}(t_1(D)) \end{aligned}$$

Por exemplo, considere a coleção D_3 que é o resultado da união entre as coleções D_1 e D_2 apresentadas anteriormente ($D_3 = \text{union}(D_1, D_2)$). Então, $\text{distinct}(D_3) = \{1, 2, 3, 4, 5\}$.

Para a definição das operações de intersecção e diferença, definimos, primeiro, as operações auxiliares *some* e *all* que representam os quantificadores existencial ($\exists$) e universal ($\forall$), respectivamente. Essas operações recebem uma função de predicado p e reduzem o conjunto de dados a um valor lógico a partir dos operadores lógicos de conjunção ($\wedge$) e disjunção ($\vee$). Para a definição de *some* e *all*, primeiro devemos mapear todos os elementos do conjunto para valores lógicos aplicando p a cada elemento de D ($t_1(p, D) = \text{flatmap}(\lambda x. \{p(x)\}, D)$). A função p define um predicado sobre um elemento x do conjunto, de modo que o seu resultado é verdadeiro (*true*) caso x satisfaça p e falso (*false*) em caso contrário. Possuindo um conjunto apenas com valores lógicos, podemos reduzir os valores do conjunto a um único valor lógico a partir da aplicação da operação **reduce** que recebe uma função binária e aplica esta de maneira associativa no conjunto. Uma vez que o quantificador existencial ($\exists$) é verdadeiro sempre que existir ao menos um valor no conjunto que satisfaça o predicado, definimos *some* a partir de uma redução aplicando o operador lógico de disjunção ($\vee$), que resultará em verdadeiro se existir ao menos um valor x em D para o qual $p(x) = \text{true}$ (**reduce**($\vee, t_1(p, D)$)). Já o quantificador universal ($\forall$), é falso se existir ao menos um elemento x em D para o qual $p(x) = \text{false}$, por isso, reduzimos o conjunto utilizando o operador de conjunção ($\wedge$), que só é verdadeiro se todos os elementos forem verdadeiros (**reduce**($\wedge, t_1(p, D)$)). Dessa forma, *some* e *all* são definidos como:

$$\begin{aligned}
& \text{some} :: (\alpha \rightarrow \text{boolean}) \rightarrow \text{Bag}[\alpha] \rightarrow \text{boolean} \\
& \text{some}(p, D) = \text{reduce}(\vee, t_1(p, D)) \\
& \text{all} :: (\alpha \rightarrow \text{boolean}) \rightarrow \text{Bag}[\alpha] \rightarrow \text{boolean} \\
& \text{all}(p, D) = \text{reduce}(\wedge, t_1(p, D)) \\
& t_1(p, D) = \text{flatmap}(\lambda x. \{\{p(x)\}\}, D)
\end{aligned}$$

Por exemplo, considere a função de predicado p que verifica se um número é maior ou igual a 1 ($p(x) = x \geq 1$) e a coleção $D = \{\{0, 1, 2, 3\}\}$. Então, $\text{some}(p, D) = \text{true}$, uma vez que existe ao menos um elemento em D para o qual o predicado em p é verdadeiro, e $\text{all}(p, D) = \text{false}$, uma vez que existe pelo menos um elemento para o qual p é falso.

Fazendo uso de some e all , podemos definir as transformações *intersection*, que representa a intersecção entre dois conjuntos de dados, e *subtract*, que representa a diferença entre dois conjuntos de dados. Para definir a intersecção (*intersection*) entre dois conjuntos D_x e D_y , consideramos que um elemento x de D_x está no conjunto resultante apenas se existir ao menos um elemento y em D_y para o qual $x = y$. Então, definimos *intersection* a partir de um mapeamento em D_x contendo uma expressão condicional (**if-then-else**) que tem como condição a operação some em D_y passando a função de predicado que verifica se $x = y$. Dessa forma, um valor x de D_x só entra no conjunto resultante se o valor de some para esse x for verdadeiro, o que resulta na intersecção entre D_x e D_y . A definição da diferença (*subtract*) entre dois conjuntos D_x e D_y é feita de forma semelhante, entretanto, a condição para que um elemento x de D_x entre no conjunto resultante é a que para todo elemento y em D_y , o predicado $x \neq y$ é verdadeiro. Assim sendo, também definimos *subtract* com um mapeamento e uma expressão condicional, mas a condição passada é a operação all passando a função de predicado que verifica se $x \neq y$, o que resulta em um conjunto contendo apenas valores que estão em D_x e não em D_y . A definição de *intersection* e *subtract* é feita a seguir:

$$\begin{aligned}
& \text{intersection} :: \text{Bag}[\alpha] \rightarrow \text{Bag}[\alpha] \rightarrow \text{Bag}[\alpha] \\
& \text{intersection}(D_x, D_y) = \text{flatmap}(\lambda x. \text{if } \text{some}(\lambda y. x = y, D_y) \text{ then } \{\{x\}\} \text{ else } \{\{\}\}, D_x) \\
& \text{subtract} :: \text{Bag}[\alpha] \rightarrow \text{Bag}[\alpha] \rightarrow \text{Bag}[\alpha] \\
& \text{subtract}(D_x, D_y) = \text{flatmap}(\lambda x. \text{if } \text{all}(\lambda y. x \neq y, D_y) \text{ then } \{\{x\}\} \text{ else } \{\{\}\}, D_x)
\end{aligned}$$

Para exemplificação, considere as coleções D_1 e D_2 que foram utilizadas no exemplo da transformação *union*. Então, $\text{intersection}(D_1, D_2) = \{\{2, 3\}\}$, $\text{subtract}(D_1, D_2) = \{\{1\}\}$ e $\text{subtract}(D_2, D_1) = \{\{4, 5\}\}$.

Agregação: são transformações que sintetizam os elementos de um conjunto de dados em um único elemento. Nos sistemas de processamento de Big Data estudados, observamos diferentes tipos de operações de agregação. A transformação de agregação mais geral é baseada na operação *Reduce* do modelo MapReduce. Definimos a operação sinônima *reduce* como a aplicação de uma função r sobre um conjunto de dados do tipo chave/valor $Bag[\kappa \times \alpha]$. A função r opera sobre uma chave do tipo κ e o grupo de elementos do tipo α associados com essa chave no conjunto de dados de entrada, gerando um conjunto de dados do tipo β ($\kappa \times Bag[\alpha] \rightarrow Bag[\beta]$). Para fazer a aplicação de r , primeiro precisamos agrupar os elementos do conjunto de dados de entrada a partir da chave do tipo κ . Para isso, aplicamos a operação **groupby** no conjunto de dados. O resultado do agrupamento é um conjunto chave/valor que tem o mesmo tipo da entrada de r ($Bag[\kappa \times Bag[\alpha]]$). Então, fazemos um mapeamento neste conjunto aplicando r , o que resulta em um conjunto do tipo $Bag[\beta]$, que é resultado da união de todos os resultados de r . Dessa forma, a transformação *reduce* é definida como segue:

$$\begin{aligned} reduce &:: (\kappa \times Bag[\alpha] \rightarrow Bag[\beta]) \rightarrow Bag[\kappa \times \alpha] \rightarrow Bag[\beta] \\ reduce(r, D) &= \text{flatmap}(r, \text{groupby}(D)) \end{aligned}$$

Para ilustrar a execução de *reduce*, vamos considerar a coleção $D = \{(w_1, 1), (w_2, 1), (w_1, 1), (w_3, 1), (w_1, 1), (w_2, 1)\}$ e a função r que recebe um elemento chave/valor do mesmo tipo dos elementos w_n nas tuplas em D e uma coleção de números inteiros associados com o elemento w_n e retorna uma tupla contendo w_n como chave e a soma dos números inteiros associados com ele como valor (por exemplo $r((w_1, \{1, 1, 1\})) = \{(w_1, 3)\}$). Então, $reduce(r, D) = \{(w_1, 3), (w_2, 2), (w_3, 1)\}$.

Outros tipos de transformações de agregação comuns são aquelas que aplicam operações binárias sobre os elementos de um conjunto de dados para gerar um único elemento, podendo isso ser feito em todo o conjunto de dados, gerando um único valor deste, ou em grupos de valores associados a uma chave em um conjunto de dados do tipo chave/valor. Representamos essas agregações com as transformações *aggregate*, que opera sobre todo o conjunto, e *aggregateByKey*, que opera sobre valores agrupados por chave. A transformação *aggregate* possui o mesmo comportamento da operação **reduce** da Álgebra de Monoides, de forma que uma função associativa f do tipo $\alpha \rightarrow \alpha \rightarrow \alpha$ é aplicada aos elementos de um conjunto D do tipo $Bag[\alpha]$ resultando em um único valor do tipo α . A definição de *aggregateByKey* também é definida em termos de **reduce**, mas uma vez que o resultado dela é a agregação dos elementos associados com uma chave ao invés da agregação de todos os elementos do conjunto, precisamos, primeiro, agrupar os elementos do conjunto pela chave (**groupby**(D)). Em seguida, fazemos um mapeamento utilizando a operação **flatmap** que resulta em um conjunto chave/valor em que a chave do grupo é

preservada e o valor passa a ser a redução do grupo através da aplicação de **reduce** com a função f no grupo.

$$\begin{aligned} \text{aggregate} &:: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \text{Bag}[\alpha] \rightarrow \alpha \\ \text{aggregate}(f, D) &= \text{reduce}(f, D) \end{aligned}$$

$$\begin{aligned} \text{aggregateByKey} &:: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \text{Bag}[\kappa \times \alpha] \rightarrow \text{Bag}[\kappa \times \alpha] \\ \text{aggregateByKey}(f, D) &= \text{flatMap}(\lambda(k, g). \{(k, \text{reduce}(f, g))\}, \text{groupby}(D)) \end{aligned}$$

Para exemplificar *aggregate* e *aggregateByKey*, vamos considerar a função f que recebe dois números x e y e retorna a soma dos dois ($f(x, y) = x + y$), e os conjuntos $D_1 = \{1, 2, 3, 4, 5\}$ e $D_2 = \{(w_1, 1), (w_2, 1), (w_1, 1), (w_1, 1), (w_2, 1)\}$. Então, $\text{aggregate}(f, D_1) = 15$ e $\text{aggregateByKey}(f, D_2) = \{(w_1, 3), (w_2, 2)\}$.

Junção: essas transformações representam operações de junção relacional entre dois conjuntos de dados. Existem diferentes tipos de junção, como a junção interna (*inner join*), que combina os elementos de dois conjuntos de dados com base em uma relação comum, como uma mesma chave, e junções externas (*outer join*), que permite a combinação de elementos de dois conjuntos de dados que possuem ou não uma relação em comum.

Nossa definição da transformação *innerJoin*, que representa uma junção interna, foi baseada na definição de junção apresentada em (CHLYAH et al., 2019). Essa transformação opera sobre dois conjuntos de dados do tipo chave/valor em que a chave de ambos é do mesmo tipo e resulta em um conjunto de dados do tipo chave/valor em que o valor contém elementos de ambos os conjuntos de entrada que eram associados com a mesma chave. Para definir *innerJoin*, que recebe um conjunto D_x do tipo $\text{Bag}[\kappa \times \alpha]$ e um conjunto D_y do tipo $\text{Bag}[\kappa \times \beta]$ como entrada, primeiro devemos co-agrupar os dois conjuntos através da aplicação da operação **cogroup**. Esse co-agrupamento resulta em um conjunto contendo elementos chave/valor em que o valor são dois grupos d_x e d_y , que possuem os valores de D_x e D_y , respectivamente, que foram associados com uma mesma chave k . Em seguida, devemos fazer mapeamentos para associar cada chave k com as combinações dos valores x em d_x com os valores y em d_y . Isso é feito através da aplicação de duas operações **flatMap** de forma aninhada. O primeiro **flatMap** mapeia os valores x em d_x , de forma que para cada x é aplicado um segundo **flatMap** que mapeia os valores y em d_y . O resultado desse segundo mapeamento é um valor da junção $(k, (x, y))$. A definição de *innerJoin* é como segue:

$$\begin{aligned}
innerJoin &:: Bag[\kappa \times \alpha] \rightarrow Bag[\kappa \times \beta] \rightarrow Bag[\kappa \times (\alpha \times \beta)] \\
innerJoin(D_x, D_y) &= \mathbf{flatMap}(\lambda(k, (d_x, d_y)).t_2(k, d_x, d_y), t_1(D_x, D_y)) \\
t_1(D_x, D_y) &= \mathbf{cogroup}(D_x, D_y) \\
t_2(k, d_x, d_y) &= \mathbf{flatMap}(\lambda x.t_3(k, x, d_y), d_x) \\
t_3(k, x, d_y) &= \mathbf{flatMap}(\lambda y.\{\{(k, (x, y))\}\}, d_y)
\end{aligned}$$

Para exemplificar a aplicação da transformação *innerJoin*, vamos considerar os conjuntos $D_1 = \{(k_1, 1), (k_2, 2), (k_3, 3), (k_4, 4)\}$ e $D_2 = \{(k_1, a), (k_2, b), (k_3, c), (k_5, e)\}$. Então, $innerJoin(D_1, D_2) = \{(k_1, (1, a)), (k_2, (2, b)), (k_3, (3, c))\}$.

As junções externas são junções entre dois conjuntos que podem conter elementos (chave/valor) de um dos dois conjuntos ou de ambos que não são associados com elementos do outro. Definimos três transformações de junção para representar junções externas, as transformações *leftOuterJoin*, *rightOuterJoin* e *fullOuterJoin*. A transformação *leftOuterJoin* representa a junção que pode conter elementos de ambos os conjuntos associados a uma mesma chave k ou elementos do primeiro conjunto D_x (*left*) que não tiveram suas chaves associadas a nenhum elemento do segundo conjunto D_y . A definição de *leftOuterJoin* é semelhante com a definição de *innerJoin* em seus primeiros passos, que consiste em co-agrupar os conjuntos D_x e D_y e em seguida fazer um mapeamento em seus valores $(k, (d_x, d_y))$. A mudança em seguida ocorre pelo fato de que mesmo quando não existir nenhum valor em d_y , o que significa que não existia em D_y valores associados com k , os valores em d_x precisam estar no resultado da junção. Dessa forma, utilizamos uma expressão condicional para verificar se d_y não possui elementos ($d_y = \{\{\}\}$). Quando isso é verdadeiro, o resultado da junção é um mapeamento de x em d_x que resulta em um valor $(k, (x, \{\{\}\}))$, de forma que o segundo elemento é representado por uma coleção vazia ($\{\{\}\}$). Em caso de d_y não ser vazio, ocorre a aplicação de duas operações **flatMap** aninhadas, semelhantes às feitas em *innerJoin*, para gerar os valores $(k, (x, \{\{y\}\}))$ da junção. A definição de *leftOuterJoin* é como segue:

$$\begin{aligned}
leftOuterJoin &:: Bag[\kappa \times \alpha] \rightarrow Bag[\kappa \times \beta] \rightarrow Bag[\kappa \times (\alpha \times Bag[\beta])] \\
leftOuterJoin(D_x, D_y) &= \mathbf{flatMap}(\lambda(k, (d_x, d_y)).t_2(k, d_x, d_y), t_1(D_x, D_y)) \\
t_1(D_x, D_y) &= \mathbf{cogroup}(D_x, D_y) \\
t_2(k, d_x, d_y) &= \mathbf{if } d_y = \{\{\}\} \mathbf{ then } t_3(k, d_x) \mathbf{ else } t_4(k, d_x, d_y) \\
t_3(k, d_x) &= \mathbf{flatMap}(\lambda x.\{\{(k, (x, \{\{\}\}))\}\}, d_x) \\
t_4(k, d_x, d_y) &= \mathbf{flatMap}(\lambda x.t_5(k, x, d_y), d_x) \\
t_5(k, x, d_y) &= \mathbf{flatMap}(\lambda y.\{\{(k, (x, \{\{y\}\}))\}\}, d_y)
\end{aligned}$$

A transformação *rightOuterJoin* representa a junção que pode conter elementos associados a uma mesma chave em ambos os conjuntos D_x e D_y ou apenas elementos com chaves presentes no segundo conjunto D_y (*right*). A sua definição é análoga à definição de *leftOuterJoin*, entretanto, ao invés de verificar se d_y é vazio, verificamos se d_x é vazio. Quando verdadeiro, é feito um mapeamento dos elementos y em d_y que resultam no valor $(k, (\{\!\!\{\!\!\}, y))$ da junção. Em caso contrário, é feita a mesma aplicação de duas operações **flatmap** aninhadas para gerar os valores $(k, (\{\!\!x\!\!\}, y))$ da junção. A transformação *rightOuterJoin* é definida como segue:

$$\begin{aligned}
& \textit{rightOuterJoin} :: \textit{Bag}[\kappa \times \alpha] \rightarrow \textit{Bag}[\kappa \times \beta] \rightarrow \textit{Bag}[\kappa \times (\textit{Bag}[\alpha] \times \beta)] \\
& \textit{rightOuterJoin}(D_x, D_y) = \mathbf{flatmap}(\lambda(k, (d_x, d_y)).t_2(k, d_x, d_y), t_1(D_x, D_y)) \\
& \quad t_1(D_x, D_y) = \mathbf{cogroup}(D_x, D_y) \\
& \quad t_2(k, d_x, d_y) = \mathbf{if} \ d_x = \{\!\!\{\!\!\} \ \mathbf{then} \ t_3(k, d_y) \ \mathbf{else} \ t_4(k, d_x, d_y) \\
& \quad \quad t_3(k, d_y) = \mathbf{flatmap}(\lambda y. \{\!\!(k, (\{\!\!\{\!\!\}, y))\!\!\}, d_y) \\
& \quad t_4(k, d_x, d_y) = \mathbf{flatmap}(\lambda x. t_5(k, x, d_y), d_x) \\
& \quad \quad t_5(k, x, d_y) = \mathbf{flatmap}(\lambda y. \{\!\!(k, (\{\!\!x\!\!\}, y))\!\!\}, d_y)
\end{aligned}$$

Por último, a transformação *fullOuterJoin* representa a junção mais geral que pode conter elementos associados em ambos os conjuntos, assim como elementos que não foram associados do primeiro e do segundo. A definição de *fullOuterJoin* também segue os dois primeiros passos de *innerJoin*. Em seguida, fazemos verificações condicionais das três possíveis combinações de d_x e d_y em expressões condicionais aninhadas, que são o caso de d_x não ser vazio e d_y ser vazio ($d_x \neq \{\!\!\{\!\!\} \wedge d_y = \{\!\!\{\!\!\}$), o caso de d_x ser vazio e d_y não ser vazio ($d_x = \{\!\!\{\!\!\} \wedge d_y \neq \{\!\!\{\!\!\}$), e o caso de ambos não serem vazios. Quando $d_x \neq \{\!\!\{\!\!\} \wedge d_y = \{\!\!\{\!\!\}$, fazemos um mapeamento de x em d_x para gerar os valores $(k, (\{\!\!x\!\!\}, \{\!\!\{\!\!\}))$ da junção. Quando $d_x = \{\!\!\{\!\!\} \wedge d_y \neq \{\!\!\{\!\!\}$, fazemos um mapeamento de y em d_y para gerar os valores $(k, (\{\!\!\{\!\!\}, \{\!\!y\!\!\}))$ da junção. Por último, quando d_x e d_y não são vazios, aplicamos duas operações **flatmap** aninhadas para gerar os valores $(k, (\{\!\!x\!\!\}, \{\!\!y\!\!\}))$ da junção. A definição de *fullOuterJoin* é como segue:

$$\begin{aligned}
fullOuterJoin &:: Bag[\kappa \times \alpha] \rightarrow Bag[\kappa \times \beta] \rightarrow Bag[\kappa \times (Bag[\alpha] \times Bag[\beta])] \\
fullOuterJoin(D_x, D_y) &= \mathbf{flatmap}(\lambda(k, (d_x, d_y)).t_2(k, d_x, d_y), t_1(D_x, D_y)) \\
t_1(D_x, D_y) &= \mathbf{cogroup}(D_x, D_y) \\
t_2(k, d_x, d_y) &= \mathbf{if } d_x \neq \{\!\!\{\!\!\} \wedge d_y = \{\!\!\{\!\!\} \mathbf{ then } t_3(k, d_x) \mathbf{ else } t_4(k, d_x, d_y) \\
t_3(k, d_x) &= \mathbf{flatmap}(\lambda x. \{\!\!\{k, (\{x\}, \{\!\!\{\!\!\})\}\!\!\}, d_x) \\
t_4(k, d_x, d_y) &= \mathbf{if } d_x = \{\!\!\{\!\!\} \wedge d_y \neq \{\!\!\{\!\!\} \mathbf{ then } t_5(k, d_y) \mathbf{ else } t_6(k, d_x, d_y) \\
t_5(k, d_y) &= \mathbf{flatmap}(\lambda y. \{\!\!\{k, (\{\!\!\{\!\!\}, \{y\})\}\!\!\}, d_y) \\
t_6(k, d_x, d_y) &= \mathbf{flatmap}(\lambda x. t_7(k, x, d_y), d_x) \\
t_7(k, x, d_y) &= \mathbf{flatmap}(\lambda y. \{\!\!\{k, (\{x\}, \{y\})\}\!\!\}, d_y)
\end{aligned}$$

Para exemplificar a aplicação de *leftOuterJoin*, *rightOuterJoin* e *fullOuterJoin*, vamos considerar os mesmos conjuntos $D_1 = \{(k_1, 1), (k_2, 2), (k_3, 3), (k_4, 4)\}$ e $D_2 = \{(k_1, a), (k_2, b), (k_3, c), (k_5, e)\}$ apresentados no exemplo da transformação *innerJoin*. Dessa forma, os resultados da aplicação de *leftOuterJoin*, *rightOuterJoin* e *fullOuterJoin* nesses dois conjuntos são:

$$\begin{aligned}
leftOuterJoin(D_1, D_2) &= \{(k_1, (1, \{a\})), (k_2, (2, \{b\})), (k_3, (3, \{c\})), (k_4, (4, \{\!\!\{\!\!\}))\} \\
rightOuterJoin(D_1, D_2) &= \{(k_1, (\{1\}, a)), (k_2, (\{2\}, b)), (k_3, (\{3\}, c)), (k_5, (\{\!\!\{\!\!\}, e))\} \\
fullOuterJoin(D_1, D_2) &= \{(k_1, (\{1\}, \{a\})), (k_2, (\{2\}, \{b\})), (k_3, (\{3\}, \{c\})), \\
&\quad (k_4, (\{4\}, \{\!\!\{\!\!\})), (k_5, (\{\!\!\{\!\!\}, \{e\}))\}
\end{aligned}$$

Ordenação: transformam um conjunto de dados de um tipo que suporta a ordenação total $\leq$ em um conjunto de dados ordenado, que na Álgebra de Monoides é representado através de listas (*List*). As transformações de ordenação são definidas em termos da operação **orderby** da Álgebra de Monoides, que transforma uma coleção $Bag[\kappa \times \alpha]$ em uma lista $List[\kappa \times \alpha]$ ordenada pela chave do tipo κ que suporta a ordem total $\leq$. Também vamos considerar a função *inv*, que inverte a ordem total de um elemento de $\leq$ para $\geq$. Além disso, na Álgebra de Monoides a operação **flatmap** preserva a ordem de elementos em uma lista, enquanto as operações **groupby** e **cogroup** não.

Dessa forma, definimos duas transformações de ordenação, a transformação *orderBy*, que ordena um conjunto de dados $Bag[\alpha]$ em que o tipo α suporta ordem total $\leq$, e a transformação *orderByKey*, que ordena um conjunto de dados do tipo chave/valor $Bag[\kappa \times \alpha]$ em que o tipo κ suporta a ordem total $\leq$. Em ambas as transformações, é possível indicar se a ordenação é feita de maneira ascendente ($\leq$), que é a ordenação padrão, ou descendente ($\geq$), a partir de um valor lógico que indica se a ordenação vai ser descendente. Para definir *orderBy*, primeiro verificamos se a ordenação vai ser descendente

ou ascendente através de uma expressão condicional (**if-then-else**). Caso a ordenação seja descendente, fazemos um mapeamento em D para transformá-lo em um conjunto de chave/valor em que a chave é a aplicação de inv no elemento, para fazer a ordenação ser descendente, e o valor é o próprio elemento. A transformação do conjunto em um conjunto chave/valor é necessária porque **orderby** opera sobre conjuntos de chave/valor. No caso em que a ordenação é ascendente, é necessário apenas fazer um mapeamento para transformar D em chave/valor em que a chave e o valor são um mesmo elemento de D . O resultado da expressão condicional, que é um conjunto do tipo $Bag[\alpha \times \alpha]$, é então aplicado na operação **orderby**, que resulta em uma lista do tipo $List[\alpha \times \alpha]$ ordenada pela chave. Por último, fazemos um mapeamento na lista para ela deixar de ser do tipo chave/valor e conter apenas as chaves do tipo α , como no conjunto inicial. Como a operação **flatMap** mantém a ordem de uma lista, a lista resultante é ordenada também. A definição de *orderBy* segue abaixo:

$$\begin{aligned}
orderBy &:: \text{boolean} \rightarrow Bag[\alpha] \rightarrow List[\alpha] \\
orderBy(desc, D) &= \text{flatMap}(\lambda(k, v).[k], \text{orderby}(t_1(desc, D))) \\
t_1(desc, D) &= \text{if } desc \text{ then } t_2(D) \text{ else } t_3(D) \\
t_2(D) &= \text{flatMap}(\lambda x. \{(inv(x), x)\}, D) \\
t_3(D) &= \text{flatMap}(\lambda x. \{(x, x)\}, D)
\end{aligned}$$

A definição da transformação *orderByKey* é semelhante à *orderBy*. Entretanto, uma vez que *orderByKey* opera sobre um conjunto do tipo chave/valor, não é necessário fazer os mapeamentos que são feitos em *orderBy* para transformar o conjunto de entrada em chave/valor. O único mapeamento que é feito ocorre quando a ordenação é feita de forma descendente, de modo que é necessário fazer um mapeamento para aplicar inv nas chaves do tipo κ para trocar a ordem total para $\geq$. Dessa forma, a transformação *orderByKey* é definida como segue:

$$\begin{aligned}
orderByKey &:: \text{boolean} \rightarrow Bag[\kappa \times \alpha] \rightarrow List[\kappa \times \alpha] \\
orderByKey(desc, D) &= \text{orderby}(t_1(desc, D)) \\
t_1(desc, D) &= \text{if } desc \text{ then } t_2(D) \text{ else } D \\
t_2(D) &= \text{flatMap}(\lambda(k, x). \{(inv(k), x)\}, D)
\end{aligned}$$

Para exemplificar as transformações de ordenação, vamos considerar o conjunto $D_1 = \{1, 3, 2, 5, 4\}$ e o conjunto $D_2 = \{(1, a), (3, c), (2, a), (5, e), (4, d)\}$. Então, a aplicação de *orderBy* e *orderByKey* nesses conjuntos seria:

$$\text{orderBy}(\text{false}, D_1) = [1, 2, 3, 4, 5]$$

$$\text{orderBy}(\text{true}, D_1) = [5, 4, 3, 2, 1]$$

$$\text{orderByKey}(\text{false}, D_2) = [(1, a), (2, b), (3, c), (4, d), (5, e)]$$

$$\text{orderByKey}(\text{true}, D_2) = [(5, e), (4, d), (3, c), (2, b), (1, a)]$$

Dentre as transformações especificadas aqui, as transformações de mapeamento e filtragem, além da transformação *union* nas transformações de conjuntos, são consideradas transformações lineares uma vez que são definidas apenas em termos de **flatMap** ou $\uplus$, que também não demanda uma reorganização. Todas as outras transformações são consideradas transformações *shuffle*. Com as transformações definidas aqui, estamos cobrindo os principais tipos de operações que são encontradas nos sistemas de processamento de Big Data estudados, como as operações existentes no Apache Spark, FlumeJava/Apache Beam, DryadLINQ e Nephele/PACTs.

6.3 Exemplo

Para exemplificar nosso modelo, vamos considerar a aplicação de contagem de frequência de palavras em um conjunto de dados que foi apresentada em cada um dos sistemas apresentados. No nosso modelo, as funções passadas como parâmetro para as transformações representam as operações sobre dados que são definidas pelo desenvolvedor. A forma em que essas funções são definidas dependem da linguagem de programação e do sistema utilizados, podendo seguir um estilo sequencial, como nos exemplos do MapReduce, Nephele/PACTs e FlumeJava/Apache Beam, ou um estilo funcional, como nos exemplos do DryadLINQ e Apache Spark. No nosso exemplo, as funções passadas para as transformações estão seguindo um estilo funcional por questão de simplicidade.

A aplicação de contagem de palavras seguiu, de maneira geral, a mesma lógica de implementação em todos os exemplos apresentados. Vamos considerar um conjunto de dados inicial (d_1) que possui os dados de texto que terão suas palavras contadas. O primeiro passo da aplicação consiste em separar cada texto em palavras individuais. Para isso, vamos utilizar a transformação *flatMap* (t_1) passando como parâmetro uma função que recebe um texto e o separa em palavras (f_1). Utilizamos a transformação *flatMap* porque para cada texto no conjunto será gerado uma coleção de palavras e as coleções geradas são unificadas em um único conjunto de dados (d_2). O passo seguinte, consiste em transformar as palavras em uma tupla chave/valor, em que a chave consiste na própria palavra e o valor o número inteiro 1. Para isso, faremos uso da transformação *map* (t_2) passando uma função que recebe uma palavra e retorna a tupla contendo a palavra e o número 1 (f_2). Essa transformação gera um conjunto de dados (d_3) contendo o mesmo número de elementos que o conjunto anterior. O último passo da aplicação consiste em

agrupar os elementos do conjunto pela chave (palavra) e somar os valores agrupados, que resultam na quantidade de vezes em que cada palavra aparece no conjunto de dados. Diferentes transformações do nosso modelo podem ser utilizadas para a definição desse último passo da aplicação, no nosso exemplo vamos utilizar a transformação *aggregateByKey* (t_3), que agrupa os valores por chave e aplica uma função associativa nos valores. A função passada para *aggregateByKey* soma os valores agrupados (f_3) e o resultado final é um conjunto de dados chave/valor em que a chave é a palavra e o valor a sua frequência (d_4). A definição da aplicação de contagem de palavra P no nosso modelo é como segue:

$$P = \langle D \cup T, E \rangle$$

O conjunto D representa os conjuntos de dados da aplicação e os conjuntos R e C são subconjuntos de D que representam os conjuntos de dados de entrada e saída, respectivamente, da aplicação:

$$D = \{d_1, d_2, d_3, d_4\}$$

$$R = \{d_1\}$$

$$C = \{d_4\}$$

$$d_1 = \text{texts}$$

$$d_2 = \text{words}$$

$$d_3 = \text{pairs}$$

$$d_4 = \text{counts}$$

O conjunto T representa as transformações aplicadas na aplicação e os conjuntos L e S são subconjuntos de T que representam, respectivamente, transformações lineares e *shuffle*:

$$T = \{t_1, t_2, t_3\}$$

$$L = \{t_1, t_2\}$$

$$S = \{t_3\}$$

$$t_1 = \text{flatMap}(f_1)$$

$$t_2 = \text{map}(f_2)$$

$$t_3 = \text{aggregateByKey}(f_3)$$

As funções passadas como parâmetro para as transformações em T são as seguintes:

$$f_1 = \text{line} \Rightarrow \text{line.split()}$$

$$f_2 = \text{word} \Rightarrow (\text{word}, 1)$$

$$f_3 = (a, b) \Rightarrow a + b$$

Por último, o conjunto E representa as arestas que ligam os conjuntos de dados e transformações da aplicação:

$$E = \{(d_1, t_1), (t_1, d_2), (d_2, t_2), (t_2, d_3), (d_3, t_3), (t_3, d_4)\}$$

7 Projetando Operadores de Mutação para Programas de Processamento de Big Data

Este capítulo apresenta um conjunto de operadores de mutação para programas de processamento de Big Data. Esses operadores foram projetados com base em duas fontes principais. A primeira é o modelo de generalização para programas de processamento de Big Data apresentado no Capítulo 6, que define uma estrutura de fluxo de dados e operações comuns para sistemas do tipo. A segunda fonte é a taxonomia de defeitos funcionais para Apache Spark apresentada no Capítulo 5, que apresenta defeitos comuns que podem aparecer no contexto de Apache Spark e que consideramos que podem ser generalizadas para sistemas de processamento de Big Data no geral.

Este capítulo está organizado da seguinte forma: na Seção 7.1 é apresentada uma classificação para os operadores de mutação propostos nesse trabalho. Na Seção 7.2 são apresentados os operadores de mutação para fluxo de dados. Na Seção 7.3 são apresentados os operadores de mutação para transformações específicas. A Seção 7.4 finaliza o capítulo com discussões sobre como os operadores de mutação propostos são relacionados com os defeitos apresentados na taxonomia de defeitos funcionais de Apache Spark e com outros tipos de operadores de mutação existentes na literatura.

7.1 Classificação dos Operadores de Mutação

Os operadores de mutação foram definidos em termos do modelo de generalização apresentado. Uma vez que o modelo é definido através de dois formalismos, um para o *fluxo de dados* e outro para as *transformações*, dividimos os operadores de mutação também em dois grupos, os *operadores de mutação para o fluxo de dados* e os *operadores de mutação para transformações*.

Os operadores de mutação para fluxo de dados definem modificações em um programa P a partir de alterações no seu fluxo de dados, que é definido através do conjunto de arestas E , que define como os conjuntos de dados em D são relacionados com as transformações em T . Dessa forma, mutações em P são definidas, geralmente, a partir de reorganização ou mudanças das arestas em E , assim como inclusão ou exclusão de transformações em T . Para esse grupo, foram definidos seis operadores de mutação que modelam defeitos relacionados ao fluxo de dados: UTS , BTS , UTR , BTR , UTD e UTI .

O segundo grupo de operadores de mutação tratam sobre modificações em trans-

formações, que são as operações sobre conjuntos de dados definidas no modelo. Esses operadores modelam defeitos em transformações específicas, podendo definir a mudança de uma transformação por outra, modificações em seus parâmetros assim como a sua inclusão ou exclusão de um programa. Para esse grupo, foram definidos 11 operadores de mutação. Os operadores de mutação de transformações definidos são: *MTR*, *FTD*, *NFTP*, *STR*, *DTD*, *DTI*, *IPOAF*, *ATR*, *JTR*, *OTI* e *OTD*. Os operadores de mutação são apresentados em detalhes nas seções a seguir.

7.2 Operadores de Mutação de Fluxo de Dados

Para a definição dos operadores de mutação de fluxo de dados, vamos considerar α , β e γ como possíveis tipos de conjuntos de dados em D . Dessa forma, vamos denominar T_α como um subconjunto de T que contem transformações unárias que possuem conjuntos de dados de entrada e saída de um mesmo tipo α , $T_{\alpha\beta}$ como um subconjunto de T que contem as transformações unárias que possuem conjuntos de dados de entrada do tipo α e conjuntos de dados de saída do tipo β , e $T_{\alpha\beta\gamma}$ como um subconjunto de T que contem as transformações binárias que recebem um conjunto do tipo α como primeiro conjunto de dados de entrada, um conjunto do tipo β como segundo conjunto de dados de entrada e produzem como saída um conjunto de dados do tipo γ :

$$T_\alpha = \{t \in T \mid input_type_1(t) = \alpha \wedge output_type(t) = \alpha\}$$

$$T_{\alpha\beta} = \{t \in T \mid input_type_1(t) = \alpha \wedge output_type(t) = \beta\}$$

$$T_{\alpha\beta\gamma} = \{t \in T \mid input_type_1(t) = \alpha \wedge input_type_2(t) = \beta \wedge output_type(t) = \gamma\}$$

UTS - Troca de Transformações Unárias (Unary Transformations Swap): se em $T_{\alpha\beta}$ existem duas transformações unárias t_1 e t_2 diferentes ($|T_{\alpha\beta}| > 1 \wedge \exists t_1, t_2 \in T_{\alpha\beta}. t_1 \neq t_2$), este operador gera um mutante ao trocar as ocorrências de t_1 por t_2 e de t_2 por t_1 , simultaneamente, em E . Por exemplo, considere os pares $(d_1, t_1) \in E$ e $(t_1, d_2) \in E$, em que d_1 e d_2 representam os conjuntos de entrada e saída de t_1 , este operador de mutação irá gerar um mutante em que esses pares são substituídos por (d_1, t_2) e (t_2, d_2) , assim como os pares $(d_3, t_2) \in E$ e $(t_2, d_4) \in E$, em que d_3 e d_4 representam os conjuntos de entrada e saída de t_2 , que são substituídos por (d_3, t_1) e (t_1, d_4) simultaneamente. Este operador gera um mutante para cada combinação possível de duas transformações diferentes em $T_{\alpha\beta}$. Dessa forma, se existirem três transformações t_1 , t_2 e t_3 diferentes em $T_{\alpha\beta}$, será gerado um mutante para os pares t_1 e t_2 , t_1 e t_3 , e t_2 e t_3 .

BTS - Troca de Transformações Binárias (Binary Transformations Swap): a definição deste operador de mutação leva em conta as transformações binárias em $T_{\alpha\beta\gamma}$. Se em $T_{\alpha\beta\gamma}$ existem duas transformações binárias t_1 e t_2 diferentes ($|T_{\alpha\beta\gamma}| > 1 \wedge \exists t_1, t_2 \in$

$T_{\alpha\beta\gamma}.t_1 \neq t_2$), este operador gera um mutante ao trocar as ocorrências de t_1 e de t_2 simultaneamente em E . Neste operador, consideramos que a ordem dos conjuntos de dados de entrada em cada transformação é mantida com a mutação. Por exemplo, considere $(d_{11}, t_1) \in E$, $(d_{21}, t_1) \in E$ e $(t_1, d_{31}) \in E$, em que d_{11} , d_{21} e d_{31} representam, respectivamente, o primeiro conjunto de dados de entrada, o segundo conjunto de dados de entrada e o conjunto de dados de saída de t_1 , o operador irá gerar um mutante em que esses pares são substituídos por (d_{11}, t_2) , (d_{21}, t_2) e (t_2, d_{31}) , assim como os pares $(d_{12}, t_2) \in E$, $(d_{22}, t_2) \in E$ e $(t_2, d_{32}) \in E$, em que d_{12} , d_{22} e d_{32} representam, respectivamente, o primeiro conjunto de dados de entrada, o segundo conjunto de dados de entrada e o conjunto de dados de saída de t_2 , serão substituídos por (d_{12}, t_1) , (d_{22}, t_1) e (t_1, d_{32}) de forma simultânea. Assim como no operador de troca de transformações unárias, para cada possível combinação de duas transformações em $T_{\alpha\beta\gamma}$, é gerado um mutante diferente.

UTR - Substituição de Transformação Unária (Unary Transformations Replacement): considerando uma transformação $t_1 \in T_{\alpha\beta}$ e $|T_{\alpha\beta}| > 1$, as ocorrências de t_1 em E são substituídas por cada uma das outras transformações em $T_{\alpha\beta}$ diferentes de t_1 ($\{t_2 \in T_{\alpha\beta} | t_1 \neq t_2\}$). Por exemplo, considere as transformações t_1 , t_2 e t_3 diferentes pertencentes a $T_{\alpha\beta}$ e os pares (d_1, t_1) e (t_1, d_2) pertencentes a E . Então, este operador irá gerar um mutante em que os pares (d_1, t_1) e (t_1, d_2) são substituídos por (d_1, t_2) e (t_2, d_2) e outro em que serão substituídos por (d_1, t_3) e (t_3, d_2) . Diferentemente do operador de troca de transformações unárias, nesse operador as modificações são feitas apenas nas ocorrências de uma transformação t específica, de forma que as ocorrências das demais transformações continuam iguais.

BTR - Substituição de Transformação Binária (Binary Transformations Replacement): considerando uma transformação binária $t_1 \in T_{\alpha\beta\gamma}$ e $|T_{\alpha\beta\gamma}| > 1$, as ocorrências de t_1 em E são substituídas por cada uma das outras transformações em $T_{\alpha\beta\gamma}$ diferentes de t_1 ($\{t_2 \in T_{\alpha\beta\gamma} | t_1 \neq t_2\}$). Por exemplo, considere as transformações t_1 , t_2 e t_3 diferentes pertencentes a $T_{\alpha\beta\gamma}$ e os pares (d_{11}, t_1) , (d_{12}, t_1) e (t_1, d_{21}) pertencentes a E . Então, será gerado um mutante em que esses pares serão substituídos por (d_{11}, t_2) , (d_{12}, t_2) e (t_2, d_{21}) e outro em que serão substituídos por (d_{11}, t_3) , (d_{12}, t_3) e (t_3, d_{21}) .

UTD - Remoção de Transformação Unária (Unary Transformation Deletion): considerando uma transformação $t \in T_{\alpha}$, este operador remove as ocorrências de t em E ao retirar os pares que contém t e ao criar um novo par que liga o conjunto que era entrada para t com as transformações que recebiam como entrada o conjunto de saída de t . Por exemplo, considerando uma transformação $t_1 \in T_{\alpha}$ e que em E existem os pares (d_1, t_1) , (t_1, d_2) e (d_2, t_2) , o operador irá gerar um mutante em que esses três pares são removidos e substituídos por (d_1, t_2) , de forma que a transformação t_1 deixa de existir no fluxo de dados.

UTI - Inserção de Transformação Unária (Unary Transformation Insertion): para este operador de mutação, vamos considerar uma transformação arbitrária t_α que possui conjuntos de entrada e saída de um mesmo tipo α ($input_type_1(t_\alpha) = \alpha \wedge output_type(t_\alpha) = \alpha$) e que não faz parte das transformações em T ($t_\alpha \notin T$). Para cada transformação $t \in T$ que possui um conjunto de dados de entrada também de um mesmo tipo α ($input_type_1(t) = \alpha$), são criadas ocorrências de t_α em E que antecedem t ao receber conjunto de dados de entrada d de t como entrada e gerar um novo conjunto de dados d_α como saída que se torna o novo conjunto de dados de entrada de t . Por exemplo, se existe o par $(d, t) \in E$, o operador irá gerar um mutante em que este par é removido e são criados os novos pares (d, t_α) , (t_α, d_α) e (d_α, t) , em que d_α é o conjunto resultante da aplicação de t_α em d .

Este operador de mutação também pode ser definido para a aplicação de t_α posterior a uma transformação t que gera como saída um conjunto de dados do tipo α ($output_type(t) = \alpha$), de modo que todos os pares que recebiam o conjunto d resultante de t são substituídos por pares que recebem d_α , que é o resultado da aplicação de t_α em d . Por exemplo, considerando as transformações $t_1 \in T$ e $t_2 \in T$, de modo que $output_type(t_1) = input_type_1(t_2) = \alpha$, e os pares $(t_1, d_1) \in E$ e $(d_1, t_2) \in E$, a aplicação deste operador em t_1 irá gerar um mutante em que este último par é removido e são criados os pares (d_1, t_α) , (t_α, d_α) e (d_α, t_2) . Uma transformação t_α pode ser definida para um programa P específico ou para um tipo de conjunto de dados α específico com o objetivo de testar certas propriedades a critério do engenheiro de testes. A aplicação de t_α de maneira anterior ou posterior a uma transformação t também fica a critério do engenheiro de testes, podendo optar por apenas uma ou ambas. Na seção seguinte, é sugerido um tipo de transformação que pode ser aplicada como t_α .

7.3 Operadores de Mutação para Transformações

Para a definição dos operadores de mutação para transformações específicas, vamos considerar uma transformação especial, denominada transformação identidade, do tipo $Bag[\alpha] \rightarrow Bag[\alpha]$ que recebe um conjunto de dados d do tipo $Bag[\alpha]$ como entrada e retorna o próprio conjunto d como saída ($identity(d) = d$).

MTR - Substituição de Transformação de Mapeamento (Mapping Transformation Replacement): este operador substitui cada aplicação de uma transformação de mapeamento ($flatMap$ e map) existente em T que recebe uma função de mapeamento f do tipo $\alpha \rightarrow \beta$ ou $\alpha \rightarrow Bag[\beta]$ por uma transformação de mapeamento que recebe uma função de mapeamento arbitrária $f_{\alpha\beta}$ do mesmo tipo de f . Uma função $f_{\alpha\beta}$ pode ser definida para uma transformação específica ou para um tipo específico a critério do engenheiro de testes. Possíveis funções $f_{\alpha\beta}$ para aplicar são funções que geram valores

contantes ou padrões do tipo adequado, além de valores incomuns para um tipo específico, por exemplo.

Essa abordagem permite, por exemplo, a aplicação de um segundo tipo de critério de teste, como os critérios de *análise do valor limite* e *classes de equivalência* (AMMANN; OFFUTT, 2017). Nesses critérios, os domínios de entrada ou saída de um programa ou função são divididos em diferentes categorias, chamadas classes de equivalência. Essas classes de equivalência representam valores válidos ou inválidos dentro do domínio, assim como valores incomuns ou que exploram limites, como um valor máximo ou mínimo aceito para um tipo específico, por exemplo. A partir dessas classes, testes são criados de modo a executar o programa ou função com valores representantes de cada uma das classes. Baseado nesses critérios, propomos alguns mapeamentos padrões para tipos comuns de modo a explorar diferentes classes de equivalência no domínio desses tipos, como os tipos numéricos (*Integer*, *Float*, *Double* e *Long*), lógicos (*Boolean*), textos (*String*) e tipos de coleções (*Collection*, *Array* e *List*). Esses tipos de dados estão presentes em todas as linguagens de programação utilizadas nos sistemas de processamento de Big Data apresentados neste trabalho, dessa forma é possível propor mapeamentos que seriam aplicáveis a todos. A Tabela 7.1 apresenta os possíveis mapeamentos, consideramos que o valor x representa a saída da função f aplicada no mapeamento.

Tabela 7.1 – Mapeamentos padrões para tipos de dados comuns.

Tipo	Mapeamentos
Numéricos	0, 1, MAX , MIN , $-x$
Lógicos	$true$, $false$, $\neg x$
Textos	“ ”
Coleções	$x.head$, $x.tail$, $x.reverse$, coleção vazia

Por exemplo, considere que em T existe a transformação $t_1 = map(f, D)$ em que $f = (x : Int) \Rightarrow x + 1$, ou seja, f mapeia um número x para o seu sucessor. Ao aplicar este operador de mutação seguindo os mapeamentos sugeridos na Tabela 7.1, seriam criados mutantes em que a função f seria alterada para funções que geram os valores 0, 1, MIN_INT (valor inteiro mínimo), MAX_INT (valor inteiro máximo) e $-(x + 1)$. Os mapeamentos propostos na Tabela 7.1 são aplicáveis apenas quando a função f mapeia um valor para um dos tipos citados. Dessa forma, quando o mapeamento é feito para algum tipo de dado diferente, a função de mapeamento a ser aplicada precisa ser definida pelo engenheiro de testes. Dada a importância de dados do tipo chave/valor, também sugerimos a aplicação desses mapeamentos para os valores da chave e/ou valor quando estes forem dos tipos citados.

FTD - Remoção de Transformação de Filtragem (Filter Transformation Deletion): remover cada aplicação da transformação *filter* existente em T . Este operador de mutação é um caso específico do operador de mutação *UTD* para fluxo de dados. Dessa

forma, a remoção de uma transformação de filtragem também implica em modificações no fluxo.

NFTP - Negação do Predicado de Transformação de Filtragem (Negation of Filter Transformation Predicate): este operador substitui cada aplicação da transformação *filter* existente em T que recebe uma função de predicado p do tipo $\alpha \rightarrow \text{boolean}$ por uma transformação *filter* que recebe uma função que nega o resultado de p ($\neg p(x)$). Por exemplo, considerando que em T existe a transformação $t_1 = \text{filter}(x \Rightarrow x > 1)$, este operador irá gerar o mutante $t_1 = \text{filter}(x \Rightarrow !(x > 1))$.

STR - Substituição de Transformação de Conjuntos (Set Transformation Replacement): substitua cada ocorrência de uma transformação de conjuntos (*union*, *intersection* e *subtract*) em T por cada uma das outras transformações de conjuntos. Além disso, substitua cada ocorrência de uma transformação de conjuntos em T pelos operadores de mutação especiais *leftDataset*, *rightDataset* e *swapDatasets*. Considerando uma transformação binária t que recebe dois conjuntos de dados d_1 e d_2 , nesta ordem, de um mesmo tipo α como entrada e retorna um conjunto de dados d_3 também do tipo α como saída (que é o caso das transformações *union*, *intersection* e *subtract*), a aplicação do operador de mutação *leftDataset* em t gera a transformação identidade de d_1 , a aplicação do operador de mutação *rightDataset* em t gera a transformação identidade de d_2 , e a aplicação do operador de mutação *swapDatasets* gera a mesma transformação t mas com a ordem dos conjuntos d_1 e d_2 invertida. Por exemplo, considerando que em T existe a transformação $t_1 = \text{union}(d_1, d_2)$, este operador de mutação irá gerar os mutantes:

$$t_1 = \text{intersection}(d_1, d_2)$$

$$t_1 = \text{subtract}(d_1, d_2)$$

$$t_1 = \text{identity}(d_1)$$

$$t_1 = \text{identity}(d_2)$$

$$t_1 = \text{union}(d_2, d_1)$$

DTD - Remoção de Transformação Distinct (Distinct Transformation Deletion): remover cada aplicação da transformação *distinct* existente em T . Este operador de mutação também é um caso específico do operador de mutação *UTD* para fluxo de dados.

DTI - Inserção de Transformação Distinct (Distinct Transformation Insertion): inserir uma aplicação da transformação *distinct* antes/após a chamada de cada uma das transformações unárias em T . A transformação *distinct* é uma transformação específica que se aplica ao operador de mutação *UTI* para fluxo de dados. Dessa forma,

inserir a transformação *distinct* em um programa P implica em modificações no fluxo em E .

IPOAF - Inversão da Ordem dos Parâmetros da Função de Agregação (Inversion of the Parameters Order of Aggregation Function): este operador substitui cada aplicação de uma transformação de agregação (*aggregate* e *aggregateByKey*) existente em T , que recebe uma função de agregação f do tipo $\alpha \rightarrow \alpha \rightarrow \alpha$, por uma transformação de agregação que recebe uma função f_2 do mesmo tipo, de modo que f_2 faz a aplicação de f passando os parâmetros de entrada com a ordem invertida. Por exemplo, considerando que em T existe a transformação $t_1 = \text{aggregate}(f, D)$ em que $f = (a : \text{String}, b : \text{String}) \Rightarrow a + b$, este operador irá gerar o mutante $t_1 = \text{aggregate}(f_2, D)$ em que $f_2 = (a : \text{String}, b : \text{String}) \Rightarrow f(b, a)$.

ATR - Substituição da Transformação de Agregação (Aggregation Transformation Replacement): este operador substitui cada aplicação de uma transformação de agregação (*aggregate* e *aggregateByKey*) existente em T , que recebe uma função de agregação f do tipo $\alpha \rightarrow \alpha \rightarrow \alpha$, por uma transformação de agregação que recebe uma função arbitrária f_α do mesmo tipo. Uma função f_α pode ser definida para uma transformação específica ou para um tipo específico a critério do engenheiro de testes. Possíveis funções f_α para aplicar são, por exemplo, funções de agregação comuns para tipos numéricos, como as funções de máximo, mínimo e soma, e funções de concatenação de coleções e textos. Por exemplo, considere que em T existe a transformação $t_1 = \text{aggregate}(f, D)$ em que f é do tipo $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$, este operador iria gerar o mutante $t_1 = \text{aggregate}(f_\alpha, D)$ em que f_α poderia ser, entre outras, uma das seguintes funções:

$$\begin{aligned} \text{max} &= (a : \text{Int}, b : \text{Int}) \Rightarrow \text{if } a > b \text{ then } a \text{ else } b \\ \text{min} &= (a : \text{Int}, b : \text{Int}) \Rightarrow \text{if } a < b \text{ then } a \text{ else } b \\ \text{first} &= (a : \text{Int}, b : \text{Int}) \Rightarrow a \\ \text{second} &= (a : \text{Int}, b : \text{Int}) \Rightarrow b \\ \text{sum} &= (a : \text{Int}, b : \text{Int}) \Rightarrow a + b \end{aligned}$$

JTR - Substituição de Transformação de Junção (Join Transformation Replacement): substitua cada ocorrência de uma transformação de junção (*innerJoin*, *leftOuterJoin*, *rightOuterJoin* e *fullOuterJoin*) existente em T por cada uma das outras transformações de junção. Por exemplo, considere que em T existe a transformação $t_1 = \text{fullOuterJoin}(d_1, d_2)$, este operador de mutação irá gerar os seguintes mutantes:

$$t_1 = \text{innerJoin}(d_1, d_2)$$

$$t_1 = \text{leftOuterJoin}(d_1, d_2)$$

$$t_1 = \text{rightOuterJoin}(d_1, d_2)$$

OTI - Inversão de Transformação de Ordenação (Order Transformation Inversion): substitua cada transformação de ordenação (*orderBy* e *orderByKey*) existente em T por uma transformação de ordenação com a ordem invertida (ascendente ou descendente). Por exemplo, considerando que em T existe a transformação $t_1 = \text{orderByKey}(\text{true}, D)$ que ordena o conjunto D em ordem descendente, o operador de mutação irá gerar o mutante $t_1 = \text{orderByKey}(\text{false}, D)$.

OTD - Remoção de Transformação de Ordenação (Order Transformation Deletion): remover cada aplicação de uma transformação de ordenação (*orderBy* e *orderByKey*) existente em T . Este operador de mutação também é um caso específico para o operador de mutação *UTD* para fluxo de dados.

7.4 Discussões

Os operadores de mutação apresentados foram definidos com o intuito de refletir defeitos que podem ocorrer durante o desenvolvimento de programas de processamento de Big Data. Para essa definição, tomamos como base a taxonomia de defeitos funcionais de Apache Spark, a estrutura de programas de processamento de Big Data definida através do nosso modelo de generalização e a inspiração em operadores de mutação clássicos que consideramos que poderiam ser aplicados nesse contexto.

Os operadores de mutação para fluxo de dados estão relacionados com o primeiro grupo ($F1$) e primeiro defeito deste grupo ($F1.1$) da taxonomia de defeitos funcionais. Este defeito trata da aplicação de transformações em uma ordem incorreta na aplicação, representando um defeito no fluxo de dados. Dessa forma, os operadores *UTS*, *BTS*, *UTR* e *BTR* foram definidos de modo a simular defeitos na sequência de transformações aplicadas em um programa através de modificações no seu fluxo de dados. Os operadores de mutação *UTD* e *UTI* foram definidos para representar interferências gerais no fluxo de dados de um programa a partir da exclusão ou inclusão de uma transformação no programa. Operadores de mutação de inclusão e exclusão de operações e declarações são clássicos em teste de mutação, podendo ser encontrados em diferentes trabalhos como (RICHARD et al., 1989), (DELAMARO; MALDONADO; MATHUR, 2001), (MA et al., 2002), (DELAMARO; OFFUTT; AMMANN, 2014) e (FERRARI; MALDONADO; RASHID, 2008).

Os operadores de mutação para transformações são relacionadas com os defeitos em transformações (*F1*) e ações (*F2*) de Spark. O operador *MTR* está relacionado ao defeitos do grupo *F1.3* que tratam sobre defeitos em transformações de mapeamento de Spark. Esses defeitos descrevem uma definição incorreta das transformações de mapeamento. Esse defeito pode ser simulado a partir de modificações na função de mapeamento passadas como parâmetros para a transformação. Propomos algumas modificações que podem ser aplicadas nos casos em que o mapeamento é feito para alguns tipos de dados comuns que estão presentes nas linguagens de programação utilizadas pelos sistemas apresentados no Capítulo 3. Essas modificações foram inspiradas em operadores de mutação para substituição de valores constantes e variáveis em um programa, como os operadores apresentados em (RICHARD et al., 1989). Os critérios de teste de *análise do valor limite* e *classes de equivalência* (AMMANN; OFFUTT, 2017) também serviram de inspiração para a definição das constantes aplicadas nessas mutações.

Os operadores de mutação *FTD* e *NFTP* estão relacionados com o grupo *F1.4* da taxonomia, que aborda defeitos em transformações de filtragem. O operador *FTD* é um caso específico para a operador *UTD* e simula um defeito de ausência de uma transformação de filtragem, assim como um defeito na sua função de predicado quando esta for definida como uma tautologia, ou seja, seu resultado é sempre verdadeiro para qualquer valor, o que implica que nenhum elemento está sendo retirado do conjunto na filtragem. O operador *NFTP* simula um defeito na função de predicado ao fazer uma negação do seu resultado, o que implica na remoção dos elementos que satisfazem o predicado ao invés do contrário. Esse tipo de operador de negação lógica também é clássico em teste de mutação, podendo ser encontrado em (AMMANN; OFFUTT, 2017) e (RICHARD et al., 1989).

Os operadores de mutação *STR*, *DTD* e *DTI* estão relacionados com os defeitos de transformações de conjuntos (*F1.2*). O operador *STR* simula a aplicação de uma transformação de conjuntos incorreta como descrita no defeito *F1.2.2*. Já os operadores *DTD* e *DTI* estão relacionados com o defeito *F1.2.1* que aborda o problema de elementos duplicados em transformações de conjuntos do Apache Spark. Uma vez que essas transformações não possuem o comportamento de operações de conjuntos matemáticos, se faz necessário chamar a transformação *distinct* para retirar elementos repetidos quando essa propriedade é esperada. Dessa forma, a inclusão ou exclusão da transformação *distinct* simula defeitos desse tipo.

Defeitos relacionados com transformações e ações de agregação (grupos *F1.5* e *F2.2*) foram representados através dos operadores de mutação *IPOAF* e *ATR*. Os defeitos *F1.5.1* e *F2.2.1* falam sobre a definição de agregações não-associativas e não-comutativas. Essas propriedades são importantes em um ambiente de programação distribuída porque definem se o resultado da agregação vai ser determinístico ou não (CHEN et al., 2017).

Procuramos simular este defeito com o operador *IPOAF*, que realiza uma permutação entre os parâmetros da função de agregação passada para a transformação. Dessa forma, quando esta função for não-associativa, o resultado da transformação será diferente. Este operador gera um mutante equivalente nos casos em que a função de agregação for associativa. O defeito de agregação não-comutativa não foi representado em um operador de mutação porque a sua verificação é dependente do sistema de processamento que executa o programa, dessa forma não teríamos como simular esse defeito com uma modificação apenas na transformação. O operador *ATR* trata do defeito de definição incorreta de uma agregação. Assim como fizemos no operador *MTR* para transformações de mapeamento, nós propomos substituições da função de agregação por agregações comuns para certos tipos de dados.

O operador de mutação *JTR* é relacionado com o grupo *F1.7* da taxonomia de defeitos que trata de defeitos em transformações de junção. Este operador representa a escolha incorreta de um tipo de junção ao substituir uma transformação de junção por junções de outros tipos. Por último, os operadores de mutação *OTI* e *OTD* são relacionados com os defeitos do grupo *F1.6* da taxonomia. Esses defeitos estão relacionados com operações de ordenação. Dessa forma, definimos um operador que inverte a ordenação (*OTI*) e um operador que remove a transformação de ordenação (*OTD*).

Os operadores de mutação propostos abordam os principais defeitos definidos na taxonomia de defeitos funcionais de Apache Spark. Além disso, os operadores exploram as características e operações de programas de processamento de Big Data que foram representadas no modelo de generalização. Isto permite que, mesmo que tenha sido definido com base em um estudo sobre Apache Spark, os operadores possam ser aplicados no teste de mutação de programas no DryadLINQ, PACTs e FlumeJava/Apache Beam, que também podem ser representados através do nosso modelo. Com base nisso, consideramos que esses operadores de mutação formam uma base para o teste de mutação de programas de processamento de Big Data.

8 Considerações Finais e Próximas Atividades

Nesta proposta de tese de doutorado visamos desenvolver uma abordagem para a aplicação de teste de mutação em programas de processamento de Big Data. Pesquisas na área de teste para programas de Big Data vem ganhando espaço nos últimos anos. Entretanto, os trabalhos existentes até o momento ainda não atingiram um bom nível de maturidade, uma vez que ainda são poucos os que aplicam técnicas e critérios sistemáticos de teste (CAMARGO; VERGILIO, 2013a) e muitos ainda carecem de validação (CSALLNER; FEGARAS; LI, 2011; MORÁN; RIVA; TUYA, 2015). Além disso, as pesquisas na área tem se concentrado majoritariamente no teste de aplicações no modelo MapReduce, como mostrado em (MORÁN; RIVA; TUYA, 2019), de modo que fica uma lacuna para o teste sistemático de programas em outros modelos e sistemas, como o Apache Spark, Dryad e DryadLINQ, FlumeJava e Nephele/PACTs.

Nesse contexto, este trabalho procura preencher esta lacuna na área ao propor uma abordagem de teste de mutação geral para programas de processamento de Big Data. Optamos pelo teste de mutação devido ao fato deste ser considerado uma referência na área de teste de software e ter a diversidade de poder ser aplicado como um critério de cobertura para projetar casos de teste (AMMANN; OFFUTT, 2017) e como um critério de qualidade na avaliação de testes e outros critérios e técnicas de teste (FRANKL; WEISS; HU, 1997; OFFUTT et al., 1996; WALSH, 1985). Dessa forma, acreditamos que uma abordagem de teste de mutação focada em programas de processamento de Big Data é uma contribuição relevante para a área.

O desenvolvimento deste trabalho seguiu uma ordem tradicional em trabalhos sobre teste de mutação (DELAMARO; OFFUTT; AMMANN, 2014). Iniciamos com um estudo para caracterizar defeitos no contexto de programas de processamento de Big Data. Escolhemos como alvo de investigação o sistema Apache Spark. Esse estudo resultou em duas taxonomias, uma taxonomia de problemas de desempenho de execução e uma taxonomia de defeitos funcionais para Apache Spark. Nosso passo seguinte foi criar um modelo para representar programas de processamento de Big Data. Nosso ponto de partida foi um levantamento sobre características comuns em programas do tipo. A partir desse modelo e da taxonomia de defeitos funcionais para Apache Spark, projetamos operadores de mutação para programas de processamento de Big Data. Os operadores de mutação formam a base para a nossa abordagem de teste de mutação. Entretanto, para aplicar o teste de mutação de forma viável é essencial ter um suporte ferramental. Dessa forma, os passos seguintes deste trabalho vão na direção de automatizar a abordagem.

Além disso, também precisamos fazer uma validação da nossa abordagem ao aplicá-la e avaliá-la em experimentos.

A seguir, resumizamos as contribuições deste trabalho desenvolvidas até o momento. Em seguida, apresentamos as atividades que ainda serão realizadas para finalizar esta tese de doutorado.

8.1 Contribuições

Abaixo listamos as contribuições deste trabalho que foram desenvolvidas até o momento:

- *Taxonomia de Problemas Desempenho para Apache Spark*: desenvolvemos uma taxonomia que agrupa e caracteriza problemas que podem afetar o desempenho de execução de aplicações desenvolvidas com o Apache Spark. Os problemas descritos podem ocorrer a nível de configuração do ambiente de *cluster* que irá executar a aplicação ou em decisões de desenvolvimento da aplicação. A taxonomia foi validada através de experimentos que mostraram a influência dos problemas descritos no desempenho de execução de aplicações;
- *Taxonomia de Defeitos Funcionais para Apache Spark*: desenvolvemos uma taxonomia que agrupa e caracteriza defeitos funcionais que podem ocorrer em aplicações desenvolvidas com o Apache Spark. Os defeitos foram agrupados de acordo com o tipo de operação (transformação e ação) ou recurso (acumuladores) de Apache Spark. Os defeitos apresentados foram demonstrados através de exemplos em códigos de aplicações;
- *Modelo de Generalização para Programas de Processamento de Big Data*: desenvolvemos um modelo para generalizar programas de processamento de Big Data. O modelo foi baseado em características comuns dos sistemas Apache Spark, Dryad e DryadLINQ, FlumeJava e Apache Beam, e Nephele/PACTs;
- *Operadores de Mutação para Programas de Processamento de Big Data*: a partir da taxonomia de defeitos funcionais para Apache Spark e do modelo de generalização, projetamos operadores de mutação para programas de processamento de Big Data. Os operadores foram projetados de modo a explorar a estrutura desse tipo de programa, tomando como referência o modelo de generalização, e refletir possíveis defeitos que podem ser encontrados em programas do tipo, tomando como referência a taxonomia. Os operadores de mutação formam a base para uma abordagem de teste de mutação para programas de processamento de Big Data.

8.2 Próximas Atividades

Os operadores de mutação desenvolvidos neste trabalho já possibilitam a aplicação do teste de mutação em programas de processamento de Big Data. Entretanto, o teste de mutação se torna impraticável sem uma automatização dado o grande esforço que é necessário para gerar e executar os mutantes (DELAMARO; MALDONADO; JINO, 2017). Dessa forma, nossas próximas atividades vão na direção de automatizar a abordagem. Nosso foco inicial será no desenvolvimento de uma ferramenta para o teste de mutação de programas desenvolvidos com o Apache Spark. Essa escolha se deve (i) à experiência adquirida com Spark durante os estudos que resultaram nas taxonomias de problemas de desempenho e defeitos funcionais, e (ii) à relevância de Apache Spark, que tem se configurado como um dos principais sistemas para processamento de Big Data. Entretanto, pretendemos projetar a ferramenta para que esta seja adaptável para outros sistemas.

Além disso, os operadores de mutação propostos neste trabalho ainda carecem de validação. Dessa forma, ainda se faz necessário por os operadores em prática para (i) ver a viabilidade e efetividade dos mesmos e (ii) realizar ajustes e melhorias a partir de detalhes que só podem ser observados na prática. Nesse sentido, acreditamos que é necessário fazer uma primeira avaliação antes do desenvolvimento de uma ferramenta que automatize a abordagem, de forma que os primeiros ajustes possam ser feitos antes do desenvolvimento. Para isso, pretendemos fazer uma busca e seleção de projetos que utilizam o Apache Spark e que contenham um conjunto de testes já desenvolvidos de modo a aplicar nossa abordagem em projetos reais, além de complementar com os projetos desenvolvidos durante o nosso trabalho. Uma segunda avaliação será feita após o desenvolvimento da ferramenta, de modo a testá-la na prática e avaliar seu uso como uma ferramenta de apoio para o teste de mutação de programas de processamento de Big Data.

Dessa forma, as próximas atividades desta pesquisa são:

- *Tarefa 1 – Avaliação dos operadores de mutação propostos:* vamos fazer uma seleção de projetos que utilizam Apache Spark para fazer a aplicação dos nossos operadores de mutação. Nesta primeira avaliação, a geração e execução dos mutantes será feita de forma manual;
- *Tarefa 2 – Escrita de um artigo científico para apresentar os operadores de mutação:* com os resultados desta primeira avaliação, além das outras contribuições já desenvolvidas, pretendemos escrever um artigo científico para fazer uma primeira apresentação da nossa abordagem de teste de mutação;
- *Tarefa 3 – Projeto da ferramenta que automatiza a abordagem:* vamos fazer um

levantamento dos principais requisitos que uma ferramenta para teste de mutação precisar ter. Para isso, vamos estudar como outras ferramentas de teste de mutação foram desenvolvidas, sua arquitetura e funcionalidades. A partir desse estudo, faremos o projeto de nossa ferramenta;

- *Tarefa 4 – Desenvolvimento da ferramenta que automatiza a abordagem:* com base no projeto feito na tarefa anterior, faremos a implementação da nossa ferramenta;
- *Tarefa 5 – Experimentos finais e avaliação da ferramenta:* com a ferramenta finalizada, pretendemos realizar novos experimentos de modo a avaliar a abordagem com um suporte ferramental. Também faremos uma avaliação acerca da qualidade e custos da nossa abordagem;
- *Tarefa 6 – Escrita da tese de doutorado:* vamos escrever a versão final da tese;
- *Tarefa 7 – Escrita de um artigo científico para apresentar a ferramenta:* pretendemos escrever um artigo para apresentar a versão final da nossa abordagem e ferramenta desenvolvida. A escrita desse artigo será feita no intervalo de tempo de um mês que existe entre a entrega da tese de doutorado para a banca avaliadora e a defesa da tese.

Um cronograma estimado para essas atividades é apresentado na Figura 8.1.

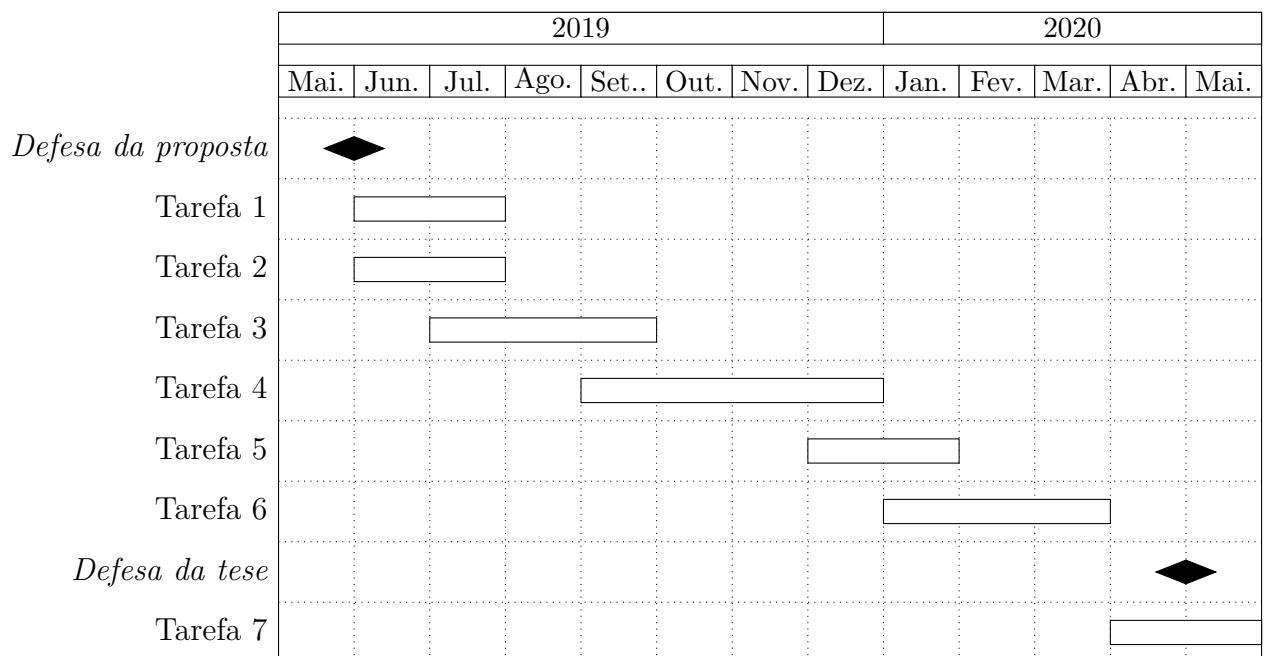


Figura 8.1 – Cronograma para as próximas atividades da pesquisa.

Referências

- ABRAN, A.; BOURQUE, P.; DUPUIS, R.; MOORE, J. W.; TRIPP, L. L. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. 2004 version. ed. Piscataway, NJ, USA: IEEE Press, 2004. 1–202 p. Disponível em: <http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf>.
- AKIDAU, T.; BALIKOV, A.; BEKIROGLU, K.; CHERNYAK, S.; HABERMAN, J.; LAX, R.; MCVEETY, S.; MILLS, D.; NORDSTROM, P.; WHITTLE, S. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In: *Very Large Data Bases*. [S.l.: s.n.], 2013. p. 734–746.
- AKIDAU, T.; BRADSHAW, R.; CHAMBERS, C.; CHERNYAK, S.; FERNÁNDEZ-MOCTEZUMA, R. J.; LAX, R.; MCVEETY, S.; MILLS, D.; PERRY, F.; SCHMIDT, E.; WHITTLE, S. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proc. VLDB Endow.*, VLDB Endowment, v. 8, n. 12, p. 1792–1803, ago. 2015. ISSN 2150-8097. Disponível em: <<http://dx.doi.org/10.14778/2824032.2824076>>.
- ALEXANDROV, A.; BERGMANN, R.; EWEN, S.; FREYTAG, J.-C.; HUESKE, F.; HEISE, A.; KAO, O.; LEICH, M.; LESER, U.; MARKL, V.; NAUMANN, F.; PETERS, M.; RHEINLÄNDER, A.; SAX, M. J.; SCHELTER, S.; HÖGER, M.; TZOUMAS, K.; WARNEKE, D. The stratosphere platform for big data analytics. *The VLDB Journal*, Springer-Verlag, Berlin, Heidelberg, v. 23, n. 6, p. 939–964, dez. 2014. ISSN 1066-8888. Disponível em: <<https://doi.org/10.1007/s00778-014-0357-y>>.
- AMMANN, P.; OFFUTT, J. *Introduction to Software Testing*. Second edition. New York, NY: Cambridge University Press, 2017.
- AMPLAB. *Big Data Benchmark*. 2019. Accessed: 2019-02-04. Disponível em: <<https://amplab.cs.berkeley.edu/benchmark/>>.
- BATTRÉ, D.; EWEN, S.; HUESKE, F.; KAO, O.; MARKL, V.; WARNEKE, D. Nephele/pacts: A programming model and execution framework for web-scale analytical processing. In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. New York, NY, USA: ACM, 2010. (SoCC '10), p. 119–130. ISBN 978-1-4503-0036-0. Disponível em: <<http://doi.acm.org/10.1145/1807128.1807148>>.
- BEAM, A. *Apache Beam: An advanced unified programming model*. 2016. Acesso em: 12 de Março de 2019. Disponível em: <<https://beam.apache.org/>>.
- BLEI, D. M.; NG, A. Y.; JORDAN, M. I. Latent dirichlet allocation. *J. Mach. Learn. Res.*, JMLR.org, v. 3, p. 993–1022, mar. 2003. ISSN 1532-4435. Disponível em: <<http://dl.acm.org/citation.cfm?id=944919.944937>>.
- BRASS, S.; GOLDBERG, C. Semantic errors in sql queries: a quite complete list. In: *Fourth International Conference on Quality Software, 2004. QSIC 2004. Proceedings*. [S.l.: s.n.], 2004. p. 250–257.

- BU, Y.; HOWE, B.; BALAZINSKA, M.; ERNST, M. D. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, VLDB Endowment, v. 3, n. 1-2, p. 285–296, set. 2010. ISSN 2150-8097. Disponível em: <<http://dx.doi.org/10.14778/1920841.1920881>>.
- CAMARGO, L. C.; VERGILIO, S. R. Classificação de Defeitos para Programas MapReduce: Resultados de um Estudo Empírico. In: *Proceedings of the 7th Brazilian Workshop on Systematic and Automated Software Testing (SAST)*. [S.l.: s.n.], 2013.
- CAMARGO, L. C.; VERGILIO, S. R. Mapreduce program testing: a systematic mapping study. In: *Chilean Computer Science Society (SCCC), 32nd International Conference of the Computation*. [S.l.: s.n.], 2013.
- CHAMBERS, C.; RANIWALA, A.; PERRY, F.; ADAMS, S.; HENRY, R.; BRADSHAW, R.; NATHAN. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2 Penn Plaza, Suite 701 New York, NY 10121-0701: [s.n.], 2010. p. 363–375. Disponível em: <<http://dl.acm.org/citation.cfm?id=1806638>>.
- CHEN, Y.-F.; HONG, C.-D.; LENGÁL, O.; MU, S.-C.; SINHA, N.; WANG, B.-Y. An executable sequential specification for spark aggregation. In: ABBADI, A. E.; GARBINATO, B. (Ed.). *Networked Systems*. Cham: Springer International Publishing, 2017. p. 421–438. ISBN 978-3-319-59647-1.
- CHILLAREGE, R.; BHANDARI, I. S.; CHAAR, J. K.; HALLIDAY, M. J.; MOEBUS, D. S.; RAY, B. K.; WONG, M. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering*, v. 18, n. 11, p. 943–956, Nov 1992. ISSN 0098-5589.
- CHLYAH, S.; GESBERT, N.; GENEVÈS, P.; LAYAÏDA, N. An Algebra with a Fixpoint Operator for Distributed Data Collections. Working paper or preprint. 2019. Disponível em: <<https://hal.inria.fr/hal-02066649>>.
- CSALLNER, C.; FEGARAS, L.; LI, C. New ideas track: Testing mapreduce-style programs. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. New York, NY, USA: ACM, 2011. (ESEC/FSE '11), p. 504–507. ISBN 978-1-4503-0443-6. Disponível em: <<http://doi.acm.org/10.1145/2025113.2025204>>.
- DEAN, J.; GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA: [s.n.], 2004. p. 137–150.
- DELAMARO, M.; MALDONADO, J.; JINO, M. *Introdução ao Teste de Software*. 2. ed. Elsevier Brasil, 2017. ISBN 9788535226348. Disponível em: <<https://books.google.com.br/books?id=7R6XPgAACAAJ>>.
- DELAMARO, M. E.; MALDONADO, J. C.; MATHUR, A. P. Interface mutation: an approach for integration testing. *IEEE Transactions on Software Engineering*, v. 27, n. 3, p. 228–247, March 2001. ISSN 0098-5589.

- DELAMARO, M. E.; OFFUTT, J.; AMMANN, P. Designing deletion mutation operators. In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. [S.l.: s.n.], 2014. p. 11–20. ISSN 2159-4848.
- DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *Computer*, v. 11, n. 4, p. 34–41, April 1978. ISSN 0018-9162.
- FEGARAS, L. An algebra for distributed big data analytics. *Journal of Functional Programming*, Cambridge University Press, v. 27, p. e27, 2017.
- FEGARAS, L. Compile-time query optimization for big data analytics. *Open Journal of Big Data (OJBD)*, RonPub, v. 5, n. 1, p. 35–61, 2019. ISSN 2365-029X. Disponível em: <https://www.ronpub.com/ojbd/OJBD_2019v5i1n02_Fegaras.html>.
- FERRARI, F. C.; MALDONADO, J. C.; RASHID, A. Mutation testing for aspect-oriented programs. In: *2008 1st International Conference on Software Testing, Verification, and Validation*. [S.l.: s.n.], 2008. p. 52–61. ISSN 2159-4848.
- FRANKL, P. G.; WEISS, S. N.; HU, C. All-uses vs mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, v. 38, n. 3, p. 235 – 253, 1997. ISSN 0164-1212. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S0164121296001549>>.
- GANELIN, I.; ORHIAN, E.; SASAKI, K.; YORK, B. *Spark: Big Data Cluster Computing in Production*. [S.l.]: Wiley, 2016. ISBN 9781119254041.
- GARG, N.; SINGLA, S.; JANGRA, S. Challenges and techniques for testing of big data. *Procedia Computer Science*, v. 85, p. 940 – 948, 2016. ISSN 1877-0509. International Conference on Computational Modelling and Security (CMS 2016). Disponível em: <<http://www.sciencedirect.com/science/article/pii/S1877050916306354>>.
- GUBBI, J.; BUYYA, R.; MARUSIC, S.; PALANISWAMI, M. Internet of things (iot): A vision, architectural elements, and future directions. *Future Gener. Comput. Syst.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 29, n. 7, p. 1645–1660, set. 2013. ISSN 0167-739X. Disponível em: <<http://dx.doi.org/10.1016/j.future.2013.01.010>>.
- GUDIPATI, M.; RAO, S.; MOHAN, N. D.; GAJJA, N. K. Big data: Testing approach to overcome quality challenges. *Big Data: Challenges and Opportunities*, p. 65–72, 2013.
- HADOOP. *Apache Hadoop Documentation*. 2019. Acesso em: 19 de Março de 2019. Disponível em: <<https://hadoop.apache.org/docs/r2.7.3/>>.
- HARPER, F. M.; KONSTAN, J. A. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, ACM, New York, NY, USA, v. 5, n. 4, p. 19:1–19:19, dez. 2015. ISSN 2160-6455.
- HASSINE, J. Design and classification of mutation operators for abstract state machines. *International Journal on Advances in Software*, v. 6, n. 1 & 2, 2013.
- HOWDEN, W. E. Weak mutation testing and completeness of test sets. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 8, n. 4, p. 371–379, jul. 1982. ISSN 0098-5589. Disponível em: <<http://dx.doi.org/10.1109/TSE.1982.235571>>.

- HUESKE, F.; PETERS, M.; SAX, M. J.; RHEINLÄNDER, A.; BERGMANN, R.; KRETTEK, A.; TZOUMAS, K. Opening the black boxes in data flow optimization. *Proc. VLDB Endow.*, VLDB Endowment, v. 5, n. 11, p. 1256–1267, jul. 2012. ISSN 2150-8097. Disponível em: <<http://dx.doi.org/10.14778/2350229.2350244>>.
- ISARD, M.; BUDIU, M.; YU, Y.; BIRRELL, A.; FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007. (EuroSys '07), p. 59–72. ISBN 978-1-59593-636-3. Disponível em: <<http://doi.acm.org/10.1145/1272996.1273005>>.
- JIA, Y.; HARMAN, M. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In: *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference*. [S.l.: s.n.], 2008. p. 94–98.
- JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, v. 37, n. 5, p. 649–678, Sep. 2011. ISSN 0098-5589.
- KALAVRI, V.; VLASSOV, V. Mapreduce: Limitations, optimizations and open issues. In: *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. [S.l.: s.n.], 2013. p. 1031–1038. ISSN 2324-898X.
- KARAU, H.; WARREN, R. *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. [S.l.]: O'Reilly Media, 2017. ISBN 9781491943175.
- KAVI, K. M.; BUCKLES, B. P.; BHAT, N. A Formal Definition of Data Flow Graph Models. *IEEE Transactions on Computers*, C-35, n. 11, p. 940–948, Nov 1986. ISSN 0018-9340.
- KUMAR, V.; ANDRADE, H.; GEDIK, B.; WU, K.-L. Deduce: At the intersection of mapreduce and stream processing. In: *Proceedings of the 13th International Conference on Extending Database Technology*. New York, NY, USA: ACM, 2010. (EDBT '10), p. 657–662. ISBN 978-1-60558-945-9. Disponível em: <<http://doi.acm.org/10.1145/1739041.1739120>>.
- LEE, K.-H.; LEE, Y.-J.; CHOI, H.; CHUNG, Y. D.; MOON, B. Parallel data processing with mapreduce: A survey. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 40, n. 4, p. 11–20, jan. 2012. ISSN 0163-5808. Disponível em: <<http://doi.acm.org/10.1145/2094114.2094118>>.
- LEE, S. C.; OFFUTT, J. Generating test cases for xml-based web component interactions using mutation analysis. In: *Proceedings 12th International Symposium on Software Reliability Engineering*. [S.l.: s.n.], 2001. p. 200–209. ISSN 1071-9458.
- LI, M.; TAN, J.; WANG, Y.; ZHANG, L.; SALAPURA, V. Sparkbench: a spark benchmarking suite characterizing large-scale in-memory data analytics. *Cluster Computing*, v. 20, n. 3, p. 2575–2589, Sep 2017. ISSN 1573-7543.
- LINQ. *Language Integrated Query (LINQ)*. 2017. Acesso em: 19 de Março de 2019. Disponível em: <<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>>.

- MA, Y.-S.; KWON, Y.-R.; ; OFFUTT, J. Inter-class mutation operators for java. In: *13th International Symposium on Software Reliability Engineering, 2002. Proceedings.* [S.l.: s.n.], 2002. p. 352–363. ISSN 1071-9458.
- MA, Y. S.; OFFUTT, J.; KWON, Y. R. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, John Wiley & Sons, Ltd., v. 15, n. 2, p. 97–133, 2005.
- MALDONADO, J. C.; DELAMARO, M. E.; FABBRI, S. C. P. F.; SIMÃO, A. da S.; SUGETA, T.; VINCENZI, A. M. R.; MASIERO, P. C. Proteum: A family of tools to support specification and program testing based on mutation. In: _____. *Mutation Testing for the New Century*. Boston, MA: Springer US, 2001. p. 113–116. ISBN 978-1-4757-5939-6. Disponível em: <https://doi.org/10.1007/978-1-4757-5939-6_19>.
- MARIANI, L. A fault taxonomy for component-based software. *Electronic Notes in Theoretical Computer Science*, v. 82, n. 6, p. 55 – 65, 2003. ISSN 1571-0661. TACoS'03, International Workshop on Test and Analysis of Component-Based Systems (Satellite Event of ETAPS 2003). Disponível em: <<http://www.sciencedirect.com/science/article/pii/S1571066104810259>>.
- MARONNA, R.; MARTIN, D.; YOHAI, V. *Robust Statistics: Theory and Methods*. [S.l.]: Wiley, 2006. (Wiley Series in Probability and Statistics). ISBN 9780470010921.
- MARR, B. *Big Data: Using SMART Big Data, Analytics and Metrics To Make Better Decisions and Improve Performance*. Wiley, 2015. ISBN 9781118965832. Disponível em: <<https://books.google.com.br/books?id=dxFTBQAAQBAJ>>.
- MEIJER, E.; BECKMAN, B.; BIERMAN, G. Linq: Reconciling object, relations and xml in the .net framework. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2006. (SIGMOD '06), p. 706–706. ISBN 1-59593-434-0. Disponível em: <<http://doi.acm.org/10.1145/1142473.1142552>>.
- MESOS. *Apache Mesos Documentation*. 2019. Acesso em: 19 de Março de 2019. Disponível em: <<http://mesos.apache.org/documentation>>.
- MITTAL, A. Article: Trustworthiness of big data. *International Journal of Computer Applications*, v. 80, n. 9, p. 35–40, 2013.
- MORÁN, J.; RIVA, C. d. l.; TUYA, J. Mrtree: Functional testing based on mapreduce's execution behaviour. In: *2014 International Conference on Future Internet of Things and Cloud*. [S.l.: s.n.], 2014. p. 379–384.
- MORÁN, J.; RIVA, C. d. l.; TUYA, J. Testing data transformations in mapreduce programs. In: *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation*. New York, NY, USA: ACM, 2015. (A-TEST 2015), p. 20–25. ISBN 978-1-4503-3813-4. Disponível em: <<http://doi.acm.org/10.1145/2804322.2804326>>.
- MORÁN, J.; RIVA, C. de la; TUYA, J. Testing mapreduce programs: A systematic mapping study. *Journal of Software: Evolution and Process*, v. 31, n. 3, p. e2120, 2019. Disponível em: <<https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2120>>.

- MURATA, T. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, v. 77, n. 4, p. 541–580, April 1989. ISSN 0018-9219.
- MYERS, G.; SANDLER, C.; BADGETT, T.; THOMAS, T. *The Art of Software Testing*. [S.l.]: Wiley, 2004. (Business Data Processing: A Wiley Series). ISBN 9780471678359.
- NACHIYAPPAN, S.; JUSTUS, S. Getting ready for bigdata testing: A practitioner's perception. In: *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*. [S.l.: s.n.], 2013. p. 1–5.
- NAKAMURA, T.; HOCHSTEIN, L.; BASILI, V. R. Identifying domain-specific defect classes using inspections and change history. In: *ACM. Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. [S.l.], 2006. p. 346–355.
- NETO, J. B. S.; MOREIRA, A. M.; VARGAS-SOLAR, G.; MUSICANTE, M. A. *Experiments of the paper “A Taxonomy of Performance Issues for Apache Spark”*. 2019. Acessado em: 02-04-2019. Disponível em: <<https://github.com/jbsneto-ppgsc-ufrn/spark-performance-issues-experiments>>.
- OFFUTT, A. J.; LEE, S. D. How strong is weak mutation? In: *Proceedings of the Symposium on Testing, Analysis, and Verification*. New York, NY, USA: ACM, 1991. (TAV4), p. 200–213. ISBN 0-89791-449-X. Disponível em: <<http://doi.acm.org/10.1145/120807.120826>>.
- OFFUTT, A. J.; PAN, J.; TEWARY, K.; ZHANG, T. An experimental evaluation of data flow and mutation testing. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 26, n. 2, p. 165–176, fev. 1996. ISSN 0038-0644. Disponível em: <[http://dx.doi.org/10.1002/\(SICI\)1097-024X\(199602\)26:2<165::AID-SPE5>3.0.CO;2-K](http://dx.doi.org/10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K)>.
- OFFUTT, J.; MA, Y.-S.; KWON, Y.-R. An experimental mutation system for java. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 29, n. 5, p. 1–4, set. 2004. ISSN 0163-5948. Disponível em: <<http://doi.acm.org/10.1145/1022494.1022537>>.
- OLSTON, C.; REED, B.; SRIVASTAVA, U.; KUMAR, R.; TOMKINS, A. Pig latin: A not-so-foreign language for data processing. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2008. (SIGMOD '08), p. 1099–1110. ISBN 978-1-60558-102-6. Disponível em: <<http://doi.acm.org/10.1145/1376616.1376726>>.
- PAVLO, A.; PAULSON, E.; RASIN, A.; ABADI, D. J.; DEWITT, D. J.; MADDEN, S.; STONEBRAKER, M. A comparison of approaches to large-scale data analysis. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2009. (SIGMOD '09), p. 165–178. ISBN 978-1-60558-551-2.
- RICHARD, H. A.; DEMILLO, R. A.; HATHAWAY, B.; HSU, W.; HSU, W.; KRAUSER, E. W.; MARTIN, R. J.; MATHUR, A. P.; SPAFFORD, E. H. *Design Of Mutant Operators For The C Programming Language*. [S.l.], 1989.
- SARWAR, B.; KARYPIS, G.; KONSTAN, J.; RIEDL, J. Item-based collaborative filtering recommendation algorithms. In: *Proceedings of the 10th International*

Conference on World Wide Web. New York, NY, USA: ACM, 2001. (WWW '01), p. 285–295. ISBN 1-58113-348-0.

SHAHRIAR, H.; ZULKERNINE, M. Music: Mutation-based sql injection vulnerability checking. In: *2008 The Eighth International Conference on Quality Software*. [S.l.: s.n.], 2008. p. 77–86. ISSN 1550-6002.

SHVACHKO, K.; KUANG, H.; RADIA, S.; CHANSLER, R. The Hadoop Distributed File System. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. [S.l.: s.n.], 2010. p. 1–10. ISSN 2160-195X.

SPARK. *Apache Spark Documentation*. 2019. Acesso em: 19 de Março de 2019. Disponível em: <<http://spark.apache.org/docs/2.2.0/>>.

THUSOO, A.; SARMA, J. S.; JAIN, N.; SHAO, Z.; CHAKKA, P.; ANTHONY, S.; LIU, H.; WYCKOFF, P.; MURTHY, R. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, VLDB Endowment, v. 2, n. 2, p. 1626–1629, ago. 2009. ISSN 2150-8097. Disponível em: <<https://doi.org/10.14778/1687553.1687609>>.

WALSH, P. J. *A Measure of Test Case Completeness (Software, Engineering)*. Tese (Doutorado) — State University of New York at Binghamton, Binghamton, NY, USA, 1985. AAI8514636.

WARNEKE, D.; KAO, O. Nephele: Efficient parallel data processing in the cloud. In: *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*. New York, NY, USA: ACM, 2009. (MTAGS '09), p. 8:1–8:10. ISBN 978-1-60558-714-1. Disponível em: <<http://doi.acm.org/10.1145/1646468.1646476>>.

YU, Y.; ISARD, M.; FETTERLY, D.; BUDIYU, M.; ERLINGSSON, U.; GUNDA, P. K.; CURREY, J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008. (OSDI'08), p. 1–14. Disponível em: <<http://dl.acm.org/citation.cfm?id=1855741.1855742>>.

ZAHARIA, M.; CHOWDHURY, M.; DAS, T.; DAVE, A.; MA, J.; MCCAULEY, M.; FRANKLIN, M. J.; SHENKER, S.; STOICA, I. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2012. (NSDI'12), p. 2–2. Disponível em: <<http://dl.acm.org/citation.cfm?id=2228298.2228301>>.

ZAHARIA, M.; CHOWDHURY, M.; FRANKLIN, M. J.; SHENKER, S.; STOICA, I. Spark: Cluster Computing with Working Sets. In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. Berkeley, CA, USA: USENIX Association, 2010. (HotCloud'10), p. 10–10. Disponível em: <<http://dl.acm.org/citation.cfm?id=1863103.1863113>>.

ZAHARIA, M.; WENDELL, P.; KONWINSKI, A.; KARAU, H. *Learning Spark*. [S.l.]: O'Reilly Media, 2015. ISBN 9781449359034.