DIAGNÓSTICO DE DESEMPENHO E RECONFIGURAÇÃO DINÂMICA EM PROCESSAMENTO DE DADOS MASSIVOS

VINÍCIUS VITOR DOS SANTOS DIAS

DIAGNÓSTICO DE DESEMPENHO E RECONFIGURAÇÃO DINÂMICA EM PROCESSAMENTO DE DADOS MASSIVOS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais – Departamento de Ciência da Computação como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Dorgival Olavo Guedes Neto

Belo Horizonte

Dezembro de 2016

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Dias, Vinícius Vitor dos Santos.

D541d Diagnóstico de desempenho e reconfiguração dinâmica em processamento de dados massivos / Vinícius Vitor dos Santos Dias. — Belo Horizonte, 2016.
xxv, 99 f. : il. ; 29cm.

Dissertação (mestrado) — Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Dorgival Olavo Guedes Neto.

1. Computação - Teses. 2. Computação de alto desempenho. 3. Balanceamento de carga. 4. Reconfiguração dinâmica. 5. Framework (Programa de computador).
I. Orientador. II. Título.

CDU 519.6*22(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Diagnóstico de desempenho e reconfiguração dinâmica em processamento de dados massivos

VINICIUS VITOR DOS SANTOS DIAS

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. DORGIVAL OLAVO GUEDES NETO - Orientador
Departamento de Ciência da Computação - UFMG

PROFA. JUSSARA MARQUES DE ALMEIDA GONÇALVES
Departamento de Ciência da Computação - UFMG

PROF. WAGNER MEIRA JÚNIOR
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 07 de dezembro de 2016.

Dedico este trabalho à minha família e aos meus amigos, pelo suporte e pelos valiosos conselhos.

Agradecimentos

Agradeço à minha mãe, pelo empenho e determinação em me proporcionar as condições propícias para que eu pudesse perseguir minhas próprias ambições. Agradeço também à minha avó, por ter sido tão presente em minha criação e me mostrar que dificuldades devem ser superadas com tranquilidade. Reconheço o esforço delas em sempre me apoiar nas minhas decisões profissionais, mesmo que isso significasse morar cada vez mais longe.

Além disso, não posso esquecer de mencionar minha família estendida do triângulo mineiro. Em Uberaba, minha terra natal, mantive poucas porém fortes amizades. Agradeço ao meu amigo/irmão Flávio, por sempre me receber bem e me incentivar a pensar sobre questões importantes da vida (sempre tive problemas com prioridades :P). Até hoje me surpreendo com a longevidade dessa amizade: 15 anos (60% da minha idade atual) e contando! Agradeço à minha amiga Gabriela pelas conversas divertidas e aleatórias. A capacidade artística dela me inspirou a exercitar minha criatividade, sempre que possível. Agradeço ao meu amigo Écio, pelas discussões práticas, filosóficas e acima de tudo, importantes para que eu tivesse uma perspectiva fora do mundo acadêmico. Com certeza, essas três pessoas foram essenciais para que minha motivação se mantivesse calibrada. Em Uberlândia, minha terra durante a graduação, fiz muitas amizades para a vida. Agradeço ao meu grupo eterno de amigos Deivisson, Marcela, Tiago, Pedro, Mateus (Nishida) e Thaína, por me motivar a almejar sempre mais e a evitar minha área de conforto.

Em Belo Horizonte, minha terra durante o mestrado, tive o privilégio de trabalhar com pessoas (eventuais amigos) brilhantes, que partilharam da mesma experiência que eu. Agradeço ao Osvaldo, da roça assim como eu, um companheiro presente em momentos tensos e tranquilos do mestrado, sempre com um senso de humor brincalhão e bem compatível com o meu. Agradeço ao Elverton, o cara mais determinado e centrado que eu já conheci, um amigo de conselhos, jogatinas e trabalhos administrativos do laboratório. Não posso esquecer do Rubens, que conheci por uma coincidência de trabalho e que se tornou um grande amigo, de conversa fácil, conhecimento variado

e conteúdo especialmente diverso. Agradeço ao Samuel, pelos cafés vespertinos, pela humildade e tranquilidade de alguém que já passou por um mestrado, pelos conselhos e pela animação em manter o pessoal do laboratório unido. Agradeço à Denise, pelas conversas psicológicas e contemplativas sobre a vida, o universo e tudo mais. Agradeço ao Paulo, por engajar o laboratório nas discussões sem sentido (*nonsense*) e pelas interessantes especulações sobre cinema e televisão. Agradeço à Camila, por me mostrar (mesmo que sem intenção) como lidar com problemas através de um humor ácido e sincero. Agradeço ao Júlio, pela sabedoria sensata e bem instrutiva, principalmente em assuntos políticos e sociais para observadores leigos como eu. Certamente, a experiência de um mestrado seria incompleta sem essa galera, obrigado a todos.

Por fim, agradeço aos professores que me guiaram neste trabalho. Em especial, ao meu professor orientador Dorgival Guedes, pela experiência e didática dignas de um verdadeiro doutor. Aprendi muito com ele ao longo dos anos de trabalho conjunto, lições valiosas sobre aspectos técnicos e responsabilidades de um profissional acadêmico. Agradeço também ao professor Wagner Meira Jr., por acompanhar o meu trabalho, por me estimular a pensar sobre problemas cada vez mais interessantes e difíceis e por me mostrar a importância da forma de um trabalho acadêmico.

"Have no fear of perfection - you'll never reach it"
(Salvador Dalí)

Resumo

O aumento crescente da quantidade de dados sendo armazenados e a variedade de técnicas propostas para suprir a demanda por processamento de cientistas de dados têm resultado em uma nova geração de ambientes e paradigmas de processamento paralelo e distribuído. Apesar desses ambientes facilitarem a tarefa de programação com abstrações de mais alto nível, obter um bom desempenho continua um desafio. Neste trabalho investigamos fatores impactantes no desempenho de aplicações típicas de processamento massivo de dados e para isso tomamos como base o ambiente Spark. Ao sistematizar a metodologia de análise sobre dimensões de diagnóstico, somos capazes de identificar cenários atípicos que deixam explícitas as limitações do ambiente e das ações comumente utilizadas para mitigação de ineficiências. Validamos nossas observações ao demonstrar o potencial de ganho em ajustes manuais de desempenho.

Finalmente, aplicamos o conjunto de lições aprendidas através do projeto e implementação de uma ferramenta extensível capaz de automatizar o processo de reconfiguração de aplicações Spark. A ferramenta utiliza como entrada logs de execuções passadas, garante a aplicação de políticas de ajuste sobre as estatísticas coletadas nos logs e leva em conta os padrões de comunicação durante a tomada de decisão. Para isso, a ferramenta identifica configurações globais que são passíveis de alteração ou pontos na aplicação do usuário onde o particionamento pode ser ajustado. Nossos resultados mostram que a ferramenta é capaz de obter ganhos de até 1,9× nos cenários considerados.

Abstract

The increasing amount of data being stored and the variety of algorithms proposed to meet processing demands of the data scientists have led to a new generation of computational environments and paradigms. These environments facilitate the task of programming through high level abstractions; however, achieving the ideal performance continues to be a challenge. In this work we investigate important factors concerning the performance of common big-data applications and consider the Spark framework as the target for our contributions. In particular, we organize our methodology of analysis based on diagnosis dimensions, which allow the identification of uncommon scenarios that provide us with valuable information about the environment's limitations and possible actions to mitigate the issues. First, we validate our observations by showing the potential that manual adjustments have for improving the applications performance.

Finally, we apply the lessons learned from the previous findings through the design and implementation of a extensible tool that automates the reconfiguration of Spark applications. Our tool leverages logs from previous executions as input, enforces configurable adjustment policies over the collected statistics and makes its decisions taking into account communication behaviors specific of the application evaluated. In order to accomplish that, the tool identifies global parameters that should be updated or points in the user program where the data partitioning can be adjusted based on those policies. Our results show gains of up to 1.9× in the scenarios considered.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

O termo Ciência dos Dados aparece como um novo paradigma para explorar e obter informação útil de processos complexos, que envolvem grandes volumes de dados. De certa forma, é uma evolução direta de disciplinas como mineração de dados e aprendizado de máquina, principalmente no que se refere aos modelos e algoritmos utilizados. Adicione-se a essa perspectiva o crescente aumento da complexidade dos dados de interesse, em dimensões como quantidade e variedade, e temos o vulgo termo *big data*, também conhecido como *dados massivos*. Com demandas cada vez maiores, cientistas de dados passam a ser exigentes quanto aos requisitos de complexidade e qualidade de suas tarefas e modelos. Assim, aplicações nesse contexto têm o potencial de serem críticas e estritas quanto ao prazo de entrega, ou seja, muitas vezes a importância do resultado final de uma análise está condicionada à eficiência do processo e por isso, a preocupação com desempenho passa a ser fundamental. Naturalmente, à medida que passamos a considerar aspectos de desempenho dos programas, uma série de compromissos emergem. No cenário descrito, talvez o principal compromisso seja o da granularidade do desempenho versus o nível de abstração no desenvolvimento de aplicações. De certa forma, esse compromisso tenta balancear uma questão clássica envolvendo as disciplinas de Compiladores e Engenharia de Software: *Como desenvolvedores de sistemas e aplicações, quanto da comodidade de programação podemos sacrificar em benefício do desempenho?*. Uma pergunta como essa levanta questões além da simples calibração do compromisso, mas inclui também discussões inevitáveis e pré-existentes sobre como avaliar de maneira apropriada tais sistemas.

O compromisso entre desempenho e abstração tem sido um fator determinante para uma tendência crescente: o desenvolvimento de ambientes que simplifiquem o modelo de execução ao ponto de facilitar a programação, o entendimento e a otimização de fatores relacionados ao desempenho. O problema é que, na prática, o efeito

introduzido por tais sistemas tem o resultado contrário ao seu propósito, em termos de desempenho. O modelo de programação *map/reduce* [Dean & Ghemawat, 2008] exemplifica bem esse cenário, onde o usuário se preocupa em descrever sua aplicação através de uma função de mapeamento e outra de redução e delega o restante dos aspectos de distribuição e paralelização para o ambiente. Naturalmente, para que isso funcione da maneira desejada, o ambiente precisa crescer em robustez e generalização. Assim, podemos dizer que há uma transferência de complexidade do programa de usuário para o ambiente de execução. Tal transferência é a principal origem do compromisso entre desempenho e nível de abstração.

Por um lado, aumentar o nível de abstração e facilitar a programação significa que a usabilidade e alcance do ambiente passam a ser maiores. De fato, cientistas e analistas de dados, que constituem os usuários mais típicos de sistemas de processamento massivo, em geral estão mais preocupados com o resultado de suas execuções (que resultam em preciosos modelos) e não diretamente com a eficiência do código utilizado na obtenção desse resultado. É papel, portanto, de pesquisadores em sistemas elaborar estratégias e ambientes de propósito geral que levem em conta aspectos de eficiência computacional.

Por outro lado, o preço a se pagar por essas vantagens está diretamente relacionado ao desempenho e flexibilidade dos ambientes. Prover abstrações de alto nível geralmente implica em adicionar camadas de abstração em diversas etapas da pilha de execução. Com isso, desenvolvedores de ambientes passam a contar com melhor manutenibilidade, reusabilidade e integração entre plataformas. Faz parte também dessa engenharia fixar um modelo de execução que não limite o desenvolvimento de nenhuma categoria de algoritmo. Uma consequência disso é que o domínio de ações para o ajuste de desempenho começa a se tornar cada vez mais limitado, pois todo plano passa a ser condicionado a um conjunto crescente de invariantes que precisam ser respeitados. Além disso, a existência de diversos níveis de abstração começam a mascarar as reais fontes de ineficiência, cuja identificação é essencial para a elaboração de estratégias concretas para a obtenção de melhor desempenho.

Plataformas como Hadoop e Spark [Zaharia et al., 2012] são implementações reais que se enquadram nesse cenário. Elas proveem abstrações de programação que simplificam a tarefa de descrição dos algoritmos, de uma forma propícia para a execução simultânea em diversas máquinas. Com base na descrição do algoritmo em alto nível, a plataforma é responsável por maximizar o desempenho das execuções e manter adequado o grau de utilização dos recursos das máquinas. Neste trabalho decidimos explorar esses compromissos tomando como base o ambiente Spark, por ser um padrão atual em processamento massivo de dados e por expor os desafios imediatamente

resultantes do cenário até aqui descrito. Destinamos a seção seguinte à discussão e contextualização desses desafios.

1.1 Desafios

Se não bastassem os desafios clássicos de programação paralela, as vantagens resultantes dos *frameworks* de processamento paralelo trazem consigo uma série de novos problemas e compromissos que influenciam diretamente o desempenho das aplicações. A partir dos programas escritos pelos usuários do sistema, *frameworks* como Hadoop e Spark quebram a computação em tarefas e estágios, sendo que os últimos delimitam pontos do fluxo de execução onde uma sincronização acontece para satisfazer dependências de dados. O paralelismo acontece dentro de cada estágio, com tarefas executando em paralelo o mesmo código, porém não existe paralelismo entre estágios dependentes.

Em especial, os ambientes de execução (ou *frameworks*) precisam lidar com três principais desafios ao buscar a melhor eficiência sem sacrificar altos níveis de abstração: *particionamento de dados*, *balanceamento de carga* e *manutenção da flexibilidade do modelo de execução*. Esses desafios refletem diretamente os esforços da comunidade em lidar com otimização de desempenho em sistema paralelos que processam dados massivos. A seguir discutimos cada um desses pontos.

Particionamento de dados e tarefas retardatárias Aplicações paralelas leem de fontes de dados distribuídas (p.ex., HDFS [Shvachko et al., 2010]) e progridem com a computação usando o número de partições de um arquivo como o primeiro grau de paralelismo a ser utilizado. Determinar o nível ideal de paralelismo pode ser extremamente complicado, pois diferentes programas podem apresentar custos variados em relação a uma mesma entrada (isto é, mesmo arquivo de dados). Além disso, as fontes variadas de dados e a complexidade do fluxo de trabalho típicos de análise de dados inviabilizam que usuários despendam tempo otimizando suas aplicações do ponto de vista do particionamento distribuído. Mais ainda, como alguns operadores dos *frameworks* exigem código do usuário, os custos de execução podem se tornar ainda mais imprevisíveis. Além de questões de uso inadequado de recursos, um particionamento ingênuo pode implicar em tarefas retardatárias, também conhecidas como *straggler tasks*. Neste momento não nos preocupamos com a definição exata de *tarefa*. Entretanto, para esta discussão inicial, é suficiente destacar que representam divisões atômicas do trabalho a ser paralelizado.

A definição de tarefas retardatárias, por si só, reflete o cenário de imprevisibilidade nos ambientes de execução: *"tarefas que tomam muito tempo em uma partição em*

relação às outras, degradando o desempenho e a utilização de recursos". Sem dúvida, tarefas com essa característica degradam o desempenho, mas essa não é a conclusão mais importante dessa frase. O ponto mais interessante é que elas indicam um problema no desempenho da aplicação em que a causa é variada e muitas vezes desconhecida. A dificuldade introduzida por tarefas retardatárias é, portanto, principalmente devido ao desafio em se diagnosticar a fonte de ineficiência. Ao longo do trabalho, detalhamos essa importância e como esse conhecimento pode auxiliar na definição de estratégias de otimização.

Balanceamento de carga O modelo de execução utilizado pode levar ao desbalanceamento e à assimetria: mesmo que recursos se tornem ociosos em alguns nós de processamento, o *framework* precisa esperar pelo término de todas as tarefas do estágio corrente antes de prosseguir com o próximo estágio. Por isso, é desejável que tarefas de um estágio estejam tão balanceadas quanto possível. Isso é difícil pois, antes de qualquer execução, há pouco conhecimento sobre a distribuição dos dados e custos dos programas podem ser dependentes de instâncias específicas da entrada sendo processada.

Manutenção da flexibilidade do modelo de execução Em muitos casos, um programa paralelo pode demandar uma mudança de planos e reconfiguração durante a execução para obter o melhor desempenho ao longo do tempo. Um modelo de execução deve ser capaz de lidar com essa dinâmica sob as perspectivas *online* e *offline*. Uma reconfiguração *online* é toda mudança de planos que acontece em tempo de execução, isto é, o sistema é capaz de detectar as fontes de ineficiência e reagir a elas ao longo de uma única execução da aplicação. Em contrapartida, uma reconfiguração *offline* diz respeito à capacidade que o sistema tem de observar o histórico de execuções passadas e com base nelas tomar decisões em eventuais re-execuções da mesma aplicação, conhecidas como *aplicações recorrentes*. Um algoritmo típico de exploração de dados pode iniciar com apenas alguns pontos, demandar pouco recurso no início de sua execução e mais tarde expandir imprevisivelmente. O sistema deve ser capaz de detectar essa elasticidade e se auto-reconfigurar de modo a utilizar uma nova quantidade de recursos adequada para o estágio atual da execução do programa. Em *frameworks* como Spark, isso se torna um grande desafio, dado que reconfigurações em tempo de execução podem implicar em reparticionamento de todo o conjunto de dados.

Assim, o desempenho das aplicações está sujeito tanto a variáveis globais da execução, como o balanceamento de carga entre estágios, quanto a aspectos locais de

escalonamento e atribuição de trabalho entre tarefas. Além disso, ao considerarmos que toda ação de ajuste de desempenho está condicionada ao ambiente de execução utilizado, algumas linhas de ação para lidar com os desafios descritos se tornam particularmente evidentes. Dessa forma, a maioria das propostas da comunidade visam adaptar o ambiente para suas aplicações típicas, ajustar as aplicações de maneira apropriada para o ambiente ou mesmo identificar gargalos de desempenho desconhecidos e menos evidentes. Na próxima seção, apresentamos os objetivos deste trabalho frente aos desafios discutidos.

1.2 Objetivos

Considerando que modelos e algoritmos para ciência de dados são irregulares e intensivos em termos de computação e comunicação, aprender como diagnosticar e lidar com gargalos de desempenho pode se tornar uma ferramenta essencial para a evolução desses ambientes. Assim, nossos objetivos com este trabalho são: *(1)* identificar fontes de ineficiência e compromissos intrínsecos e não óbvios em cargas reais de processamento massivo; *(2)* sistematizar a análise de desempenho através de dimensões de diagnóstico bem definidas; e *(3)* automatizar a aplicação do conhecimento adquirido através de projeto e implementação de uma ferramenta de reconfiguração construída sobre Spark.

1.3 Contribuições do trabalho

Enumeramos as seguintes contribuições deste trabalho:

1. **Diagnóstico de Desempenho**: apresentamos uma análise do desempenho de aplicações típicas de processamento massivo de dados. Essa análise está organizada de acordo com *dimensões de diagnóstico*, que nos permitem destacar ineficiências, discutir soluções comumente empregadas e principalmente, identificar oportunidades para melhorias.

2. **Ferramenta de Reconfiguração**: aliamos as lições obtidas a partir da estratégia de diagnóstico às demandas típicas de usuários do contexto (cientistas de dados) e então, descrevemos o projeto e a implementação de uma ferramenta de reconfiguração para aplicações Spark. A ferramenta utiliza informação de logs de execuções passadas para realizar ajustes baseados em políticas extensíveis.

1.4 Organização do texto

Adotamos uma metodologia *top-down* ao longo do trabalho. Isso nos permitiu traçar um plano lógico de trabalho construído incrementalmente. Dividimos portanto a metodologia em uma sequência de passos imediatamente dependentes entre si.

Em um primeiro momento, contextualizamos nossas contribuições junto à comunidade (capítulo 2). O segundo passo consiste em uma descrição do modelo de execução considerado, etapa primordial na definição de estratégias para lidar com os desafios descritos (capítulo 3). O terceiro passo tem o objetivo de definir a carga de trabalho para avaliação do modelo de execução e principalmente, justificar sua escolha no contexto de aplicações e seus padrões recorrentes (capítulo 4). O quarto passo considera o modelo de execução, suas limitações e a carga de trabalho escolhida para revisitar os desafios, identificando de maneira concreta gargalos de desempenho e oportunidades para otimização (capítulo 5). Finalmente, o quinto passo interpreta o conjunto de lições aprendidas e realiza um esforço na proposição de uma ferramenta para auxílio na otimização dos problemas encontrados (capítulo 6). Sendo assim, o restante do trabalho está organizado nos seguintes capítulos:

Capítulo 2: Trabalhos relacionados Descrevemos os trabalhos da comunidade necessários para a contextualização de nossos objetivos e discutimos a relevância das nossas contribuições frente a esse cenário.

Capítulo 3: Modelo de execução Descrevemos o modelo de execução considerado do ponto de vista do ambiente de computação empregado (Spark) juntamente com as dimensões de desempenho consideradas para avaliar esse modelo.

Capítulo 4: Algoritmos e padrões de comunicação Estabelecemos as características típicas de aplicações de processamento de dados massivos, categorizamos as aplicações em grupos contento características em comum e por fim, descrevemos algoritmos representativos de cada grupo para servir como fonte de carga para caracterização e validação das contribuições apresentadas.

Capítulo 5: Caracterização do desempenho Este capítulo primeiramente introduz o ambiente experimental a ser utilizado ao longo de todo o trabalho. Em um segundo momento realizamos uma caracterização das aplicações modelo de modo a: *(1)* expor as fontes de ineficiência mais comuns nesse contexto e, principalmente, *(2)* as oportuni-

dades para obtenção de ganhos em desempenho e estratégias possíveis para o proveito dessas oportunidades.

Capítulo 6: Ferramenta de Reconfiguração Apresentamos motivação, projeto e avaliação de uma ferramenta de reconfiguração que automatiza as metodologias estudadas no capítulo 5 em um contexto de aplicações recorrentes.

Capítulo 7: Conclusão Fechamos o trabalho com os destaques e trabalhos futuros mais importantes.

Capítulo 2

Trabalhos Relacionados

Os trabalhos relacionados foram organizados em duas frentes. Na primeira, composta pelas seções 2.1 e 2.2, discutimos os esforços para entender e diagnosticar características e fenômenos resultantes do modelo de execução. Essa discussão fundamenta a sistematização de qualquer proposta de sistema de ajuste de desempenho. Na segunda, composta pelas seções 2.3 e 2.4, discutimos o que tem sido proposto de concreto para otimizar o desempenho de aplicações paralelas.

2.1 Diagnóstico de desempenho

O desbalanceamento de carga em sistemas de processamento de dados massivos, como no Spark e no Hadoop, é um problema conhecido amplamente conhecido e diversas soluções foram propostas desde a popularização do modelo MapReduce [Dean & Ghemawat, 2008]. Uma alternativa muito utilizada nesse contexto é dividir aplicações complexas em diversas tarefas com granularidade muito baixa [Ousterhout et al., 2013]. Nesse caso a intuição é que tarefas pequenas são mais ajustáveis ao longo da execução e por isso levam a um balanceamento melhor, além de impactarem menos o tempo total quando uma distribuição satisfatória não puder ser feita. Outro ganho com essa abordagem acontece principalmente por se tratar de uma solução com pouca complexidade. Entretanto, o nosso estudo considera casos em que o aumento arbitrário no número de tarefas influencia pouco no desempenho ou é incapaz de capturar anomalias de desbalanceamento quando negligenciamos a distribuição do dado.

O principal trabalho que estuda sistematicamente os gargalos de desempenho em ambientes de processamento paralelo da nova geração é descrito por Ousterhout et al. [2015]. Os autores propõem uma metodologia de análise blocada ao quebrar os tempos de execução das tarefas em subintervalos, de forma a diagnosticar pontualmente as

fontes de ineficiência. Nosso trabalho considera uma abordagem diferente ao realizar uma análise experimental de baixo nível em aplicações que representam padrões de comunicação típicos em análise de dados. Assim, a abordagem adotada por Ousterhout et al. [2015] é em largura, enquanto este trabalho realiza análises em profundidade. Assim, demonstramos e validamos algumas oportunidades para refinamento de desempenho frente aos diagnósticos realizados. Uma vez que levamos em conta a natureza das aplicações durante as otimizações propostas, somos capazes de prover uma correspondência mais clara entre ineficiências e respectivos pontos de ajuste no código do usuário. Inclusive, tais características facilitam a automatização do ajuste de desempenho nas aplicações, como demonstrado através do desenvolvimento da ferramenta de reconfiguração.

Por fim, existem outros esforços recentes em relacionar o desempenho em sistemas de processamento massivo com os respectivos componentes de execução. O exemplo mais completo é o estudo feito por Shi et al. [2015], que faz uma comparação baseada em execuções de Spark e Hadoop. Os autores discutem as diferenças nos modelos de programação e componentes de cada sistema e fazem uma referência direta a isso nas análises de desempenho. Uma das implicações daquele trabalho é justamente deixar claro como os parâmetros de configuração das execuções podem impactar de forma não óbvia os módulos dos ambientes. Apesar de um de nossos objetivos ser proporcionar um melhor entendimento sobre o desempenho das aplicações, tomamos como base uma perspectiva diferente. Enquanto Shi et al. [2015] realiza uma análise profunda do ponto de vista do modelo de execução e módulos do sistema, nós destacamos a importância de se considerar também o plano de execução (DAG) como recurso recompensador durante reconfiguração das aplicações. Portanto, consideramos a proposta como ortogonal, com a diferença principal sendo a metodologia adotada.

2.2 Impacto da coleta de lixo no desempenho das aplicações

A maioria dos *frameworks* de processamento paralelo executam sobre máquinas virtuais com coleta de lixo. Esse é o caso do Spark e da Máquina Virtual Java (JVM). Diversos algoritmos de coleta de lixo sofrem de pausas globais na execução e utilização excessiva de memória em alocações. O projeto Broom [Gog et al., 2015] propõe um gerenciamento de memória refinado em regiões que alocam objetos com o mesmo padrão de acesso, evitando as excessivas varreduras de toda a memória da máquina virtual. Outros trabalhos consideram soluções holísticas [Maas et al., 2015] e argumentam que a coleção

de lixo em um ambiente distribuído deve ser feita de forma coordenada. Isso poderia auxiliar o escalonador dos ambientes a tomar melhores decisões ao difundir o estado dos trabalhadores do *cluster*. O resultado final seria uma mitigação no número de efeitos colaterais anômalos, também discutidos neste trabalho.

O gerenciamento de memória no Spark está em processo de mudança no sentido de retirar da JVM a responsabilidade de lidar com memória de partes críticas da execução [Xin, 2015], como agregações e junções. Mesmo com isso, otimizadores de desempenho ainda poderiam se beneficiar de conhecimento acerca do estado dos trabalhadores. Por fim, instrumentações internas ou externas à JVM poderiam prover um *feedback* ainda mais poderoso nesse contexto.

2.3 Otimização de Aplicações

Diversos trabalhos lidam com o problema de otimização de desempenho em *frameworks* de processamento paralelo [Chaiken et al., 2008; Ke et al., 2013; Agarwal et al., 2012]. Esses sistemas são construídos com um foco em dados estruturados (com forte apelo para o padrão SQL), onde o conhecimento sobre a organização dos registros proporciona uma tomada de decisões melhor fundamentada. Nosso trabalho considera execuções de propósito geral, em que pouca ou mesmo nenhuma premissa pode ser feita sobre os dados de entrada.

Em um *cluster* de processamento massivo, aplicações recorrentes podem ser otimizadas com o auxílio de um repositório do histórico de execuções semelhantes, como é o caso do sistema Scope [Bruno et al., 2012]. Nesse caso, a otimização de fato acontece em futuras execuções e assume que aplicações que executam o mesmo programa processam dados de entrada diferentes porém com propriedades similares. Nossas contribuições se fundamentam em um modelo de execução diferente, não assumindo nada sobre a organização dos dados, e consideram o padrão de comunicação das aplicações como fator contribuidor nas decisões tomadas.

Outras alternativas como o trabalho Ke et al. [2011] otimizam programas a partir das funções definidas pelos usuários (UDF, ou *user defined functions*) e consideram cenários tanto *online* quanto *offline*. Entretanto, os autores consideram o Dryad como ambiente de execução, que possui um modelo de representação mais flexível para refinamento de execuções devido à sua representação de DAGs propícia para re-escrita. De fato, o modelo de execução do Dryad considera como unidade gerenciável cada tarefa submetida, o que implica em uma flexibilidade maior para alterar o plano durante uma mesma execução. Nosso estudo considera um modelo de execução mais estático

(Spark), onde o particionamento deve refletir a imutabilidade dos RDDs, complicando ou inviabilizando a alteração dinâmica do plano de execução. Nesse caso, as unidades gerenciáveis refletem um conjunto de tarefas.

Ainda na questão de aplicações recorrentes, existem trabalhos que procuram melhorar o desempenho dos programas ao reconfigurar as propriedades estáticas de execução dos ambientes [Herodotou & Babu, 2011]. A ferramenta Starfish [Herodotou et al., 2011] é capaz de construir perfis de aplicações baseados em execuções passadas. Com essa informação, a cada nova submissão, ela explora o espaço de possibilidade de configurações do Hadoop com o propósito de achar o melhor conjunto de configurações para o perfil em questão. Entretanto, o Starfish considera apenas parâmetros estáticos do ambiente que precisam ser escolhido antes do início da execução. Por isso, esse aspecto também é ortogonal às otimizações baseadas na execução anterior de estágios dentro de uma mesma execução.

2.4 Execução adaptativa no Spark

Uma proposta de execução adaptativa para o Spark considera alterar a maneira com que o sistema lida com as incertezas no quesito particionamento [Zaharia, 2015]. A ideia principal é atrasar a submissão de estágios para permitir um particionamento que considere as estatísticas coletadas em estágios ancestrais. Essa nova funcionalidade, que não considera nada sobre o padrão evolutivo de comunicação das aplicações, poderia se beneficiar do conhecimento extra adquirido durante a execução de algoritmos iterativos ou mesmo re-execuções de estágios similares, pontos apresentados como contribuição deste trabalho. Por outro lado, por focar em questões de particionamento, aquele trabalho se vale de algumas das mesmas premissas desta dissertação.

Essa disposição para os trabalhos relacionados retrata a organização do texto e as contribuições. Como visto, o desenvolvimento do trabalho foi dividido em dois principais momentos. No primeiro, caracterizamos as aplicações de interesse, com o objetivo de diagnosticar fontes de ineficiência e identificar oportunidades para ajuste de desempenho. Essa contribuição estende as propostas apresentadas nas seções 2.1 e 2.2. No segundo, propomos um sistema para auxiliar o processo de ajuste de aplicações Spark de forma automática, complementando as propostas descritas nas seções 2.3 e 2.4. No capítulo seguinte, detalhamos os conceitos relacionados deste trabalho.

Capítulo 3

Modelo de execução

Para discutirmos as métricas de desempenho e a solução proposta, é preciso primeiro entender a forma como opera uma aplicação Spark. Tendo isso e mente, o *ambiente de execução* (seção 3.1) descreve como o recurso físico é organizado logicamente para comportar a execução das aplicações. Adicionalmente, o *modelo de aplicação* (seção 3.2) descreve como o trabalho das aplicações é dividido e organizado para uma execução paralelizada e distribuída. Em geral, esse modelo implica em definir a concepção de uma tarefa, suas dependências e o funcionamento do escalonador. Assim, o ambiente de execução deve ser genérico o bastante para suportar a execução de qualquer aplicação que obedeça o modelo pre-estabelecido. Do ponto de vista de desempenho, o ambiente de execução nos ajuda a avaliar as aplicações enquanto o modelo de aplicação é útil na definição do escopo de ações para o ajuste da eficiência das execuções. Neste trabalho, escolhemos o ambiente Spark como *framework* de estudo, por ser um sistema largamente utilizado pela academia e indústria, sendo capaz de expor os grandes desafios da área de processamento paralelo de dados.

Com o objetivo de obter maior clareza no entendimento do desempenho de aplicações paralelas, elaboramos um conjunto de dimensões para suportar a avaliação experimental. Como discutido, os desafios nesse contexto podem ser tão complexos e dependentes entre si que a tarefa de definir a causa de qualquer ineficiência pode não ser trivial. A nossa abordagem avalia o comportamento das aplicações em diferentes pontos de vista, isto é, considerando diferentes *dimensões de diagnóstico*. Listamos e descrevemos as dimensões e suas respectivas motivações na seção 3.3.

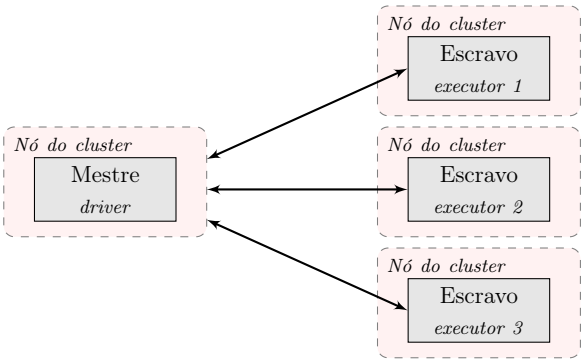


Figura 3.1: Modelo de execução de uma aplicação paralela

3.1 Ambiente de execução

Em uma infraestrutura típica para processamento de dados massivos, os recursos do *cluster* são compostos por um agregado de máquinas, virtuais ou não. Essas máquinas são organizadas fisicamente em *racks*. Nessa configuração, máquinas em um mesmo *rack* são interconectadas por uma rede sem contenção, enquanto máquinas de diferentes *racks* são interconectadas por uma inter-rede compartilhada com outros *racks*. Esse modelo pressupõe que, a qualquer momento, uma aplicação paralela pode ser instanciada em um subconjunto de máquinas do *cluster*, comumente sob uma arquitetura mestre/escravo.

A figura 3.1 representa uma aplicação mestre/escravo instanciada em três máquinas do *cluster*. Note a existência de uma entidade central (mestre da aplicação) responsável por coordenar e atribuir trabalho aos trabalhadores (escravos), que potencialmente residem em outras máquinas. No framework Spark, o mestre recebe o nome de programa *driver* e os escravos recebem o nome de *executores*. Em um modelo como esse, a comunicação acontece de forma bidirecional entre o driver e os executores.

Variáveis para ajuste de desempenho

As oportunidades para ajuste de desempenho no nível do ambiente de execução estão diretamente relacionadas com os recursos disponíveis nos sistemas operacionais que compõem o *cluster*. Recursos ofertados nesse cenário incluem processamento, memória e disco, principalmente. Em um *cluster*, esses recursos são controlados pelo escalonador de recursos do *cluster*, que tem a responsabilidade de atender a diferentes *frameworks* de processamento e/ou armazenamento. Os dois escalonadores de recursos mais utilizados no contexto de processamento massivo são o YARN [Vavilapalli et al., 2013] e o Mesos [Hindman et al., 2011]. Apesar de considerarmos alguns parâmetros relacionados com

o ambiente de execução para caracterizar aplicações, essa categoria de ajuste não é o foco do trabalho. Assim, estamos mais interessados em ajustar o desempenho de aplicações *dado um ambiente de execução*. Para isso, se faz necessária uma descrição mais profunda de como as aplicações consideradas são modeladas sobre esse ambiente, como a abstração de dados e o como as tarefas são organizadas e atribuídas pelo escalonador do *framework*. Essas questões são detalhadas a seguir.

3.2 Modelo de aplicação

Em geral, *frameworks* de processamento massivo de dados expressam computação através de procedimentos a serem aplicados aos elementos de uma coleção de dados. De fato, podemos representar um programa paralelo como um fluxo composto por operadores, entradas e saídas. Assim, com o objetivo de obter paralelismo, o dado é particionado e distribuído entre os nós de processamento.

As operações aplicadas sobre os dados em ambientes como MapReduce [Dean & Ghemawat, 2008], Dryad [Isard et al., 2007] e Spark [Zaharia et al., 2012] são descritas através de DAGs (**G**rafos **A**cíclicos e **D**irecionados). A interpretação dada aos vértices e arestas dessa estrutura dependem do sistema, entretanto sempre indicam a maneira com que o dado trafega e é transformado pelos operadores. Por exemplo, no Dryad, vértices representam tarefas individuais (isto é, um operador aplicado a uma porção do dado) e arestas representam o fluxo de entrada e saída entre as tarefas do sistema. Por outro lado, no Hadoop, os vértices representam as etapas possíveis do modelo, ou seja, um mapeamento seguido por uma redução. Adicionalmente, as arestas representam as dependências entre as duas etapas. Assim sendo, independente da representação, DAGs são essenciais para identificar oportunidades de paralelismo e necessidades de sincronização ao longo da execução.

3.2.1 Coleções de dados e operações

A máquina de execução do Spark expressa programas através de DAGs de coleções distribuídas (**R**esilient **D**istributed **D**atasets, ou RDDs). Os itens de um RDD são organizados em *partições*, que representam as unidades de computação do sistema. O processamento de uma partição é feito por uma tarefa e uma tarefa processa uma única partição de um RDD, assim, a relação entre partição e tarefa é um-para-um. A estratégia de particionamento de um RDD é determinada por um *particionador*, que pode ser baseado em *hashing*, *ranging* ou customizado pelo usuário. O conceito de partição é importante para nossa discussão porque tanto sua granularidade (grau

de paralelismo) quanto sua estratégia (particionador) influenciam diretamente como os recursos do ambiente são usados. A figura 3.2 ilustra a estrutura básica de uma coleção, nesse caso, sendo composta por n itens organizados em r partições.

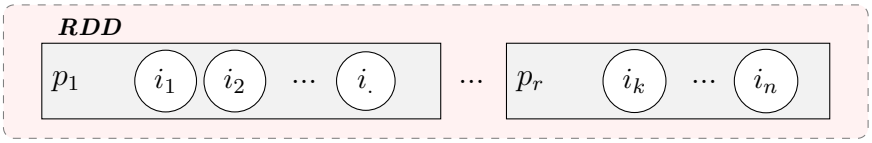


Figura 3.2: Estrutura de um RDD formado por n itens organizados em r partições

Se por um lado RDDs representam os dados de uma computação, as operações definem as dependências de dados entre eles e ditam o fluxo de execução da aplicação. Em sistemas de processamento massivo modernos, como no Spark, existem operações que exigem uma injeção de código de usuário, denominadas UDFs (***U**ser **D**efined **F**unctions*). Como essas funções são aplicadas invariavelmente em conjunto com a semântica da operação, a flexibilidade que o usuário tem de exprimir sua aplicação é maior. Existem dois tipos de operações que podem ser aplicadas sobre RDDs: *(i)* transformações e *(ii)* ações. Uma *transformação* é uma operação aplicada sobre um ou mais RDDs e cujo resultado é um novo RDD, ou seja, uma nova coleção de dados. Uma *ação* é uma operação que exige algum tipo de sincronização, como obter o número de elementos de um RDD, coletar uma amostra, salvar o conteúdo do RDD em um sistema de armazenamento externo [Shvachko et al., 2010], etc. Uma forma fácil de diferenciar esses dois tipos de operações é classificar transformação como aquilo que compõe o DAG de uma aplicação e ação como aquilo que força a computação desse DAG.

Os operadores que transformam as coleções o fazem de acordo com diferentes tipos de dependências [Zaharia et al., 2012]. Essas dependências podem ser do tipo *estreita* ou *ampla*, dependendo de como os elementos de um RDD são derivados de outro(s) RDD(s). Operações como mapeamento e filtro são os exemplos mais comuns de transformações e geram dependências do tipo *estreita*, pois novos elementos dependem apenas de um respectivo elemento do RDD ancestral. Nesse caso, a computação não requer comunicação remota, denominada *shuffle*.

Quando novas coleções são obtidas através de junções (*joins*) ou reduções por exemplo, os elementos precisam ser reorganizados e uma comunicação global entre nós de processamento se faz necessária. Situações como essa descrevem típicos cenários onde transformações geram dependências do tipo *ampla*.

3.2.2 Estágios de execução

Em Spark, cadeias de operadores com dependências *estreitas* podem ser e são agrupadas em estágios de execução através de composição de funções e *pipelining*. Como efeito desse encadeamento, RDDs gerados por transformações *estreitas* herdam o mesmo esquema de particionamento dos RDDs de origem.

Fronteiras entre estágios são determinadas pelas dependências *amplas*. A comunicação global nesses pontos representa uma oportunidade para redefinição do nível de paralelismo da aplicação, pois o particionamento pode ser ajustado para o RDD resultante da transformação. Essa oportunidade de ajuste do paralelismo representa uma dos aspectos mais efetivos no processo de reconfiguração das aplicações. Como veremos no próximo capítulo, a quantidade de partições de um RDD é de fato um dos fatores determinantes no desempenho dos programas paralelos. Por isso, damos o nome de *ponto de adaptação* às fronteiras criadas pelas dependências *amplas*, em que o particionamento pode ser alterado naturalmente e sem nenhuma violação do modelo de execução.

```
1 object Wordcount {
2   def main(args:  Array[String]) {
3      val conf = new SparkConf().setAppName("Word Count").
4          set("spark.adaptive.logpath", "/tmp/log1")
5      val sc = new SparkContext(conf)
6      val counts = sc.textFile ("/tmp/sample").
7          flatMap (_ split ).
8          map (w => (w,1)).
9          reduceByKey (_ + _)
10     println (counts.count + "words")
11     sc.stop()
12  }
13 }
```

Figura 3.3: Conta-palavras implementado em Spark

A figura 3.3 representa a implementação de uma aplicação conta-palavras em Spark. O objetivo é simples: dado um documento espalhado em diversas máquinas, encontre o número de ocorrências de cada palavra que aparece no texto. Para isso, assumimos uma coleção inicial distribuída de linhas do documento (*linha 6*), que é transformada em uma nova coleção de palavras (*linha 7*). Cada palavra é então transformada num par `(palavra, 1)`, gerando uma coleção de pares (*linha 8*). Os pares são então agrupados por chave e os valores são somados de modo a representar a frequência de cada chave (palavra) (*linha 9*).

Todo programa escrito em Spark denota uma divisão lógica da aplicação em estágios de execução (figura 3.4). Em nossa descrição do conta-palavras, mencionamos coleções de dados distribuídas sendo obtidas a partir de outras. Essas coleções são exatamente os RDDs em Spark. Como as operações *a*, *b* e *c* geram dependências do tipo *estreita*, as mesmas puderam ser agrupadas em um mesmo estágio (Estágio I). Os retângulos `linhas`, `palavras` e `pares` representam, respectivamente, os RDDs produzidos nas linhas 6, 7, e 8 da figura 3.3. Por outro lado, a operação *c* representa uma operação que, por definição, implica em comunicação global e remota entre os trabalhadores (dependência *ampla*). O resultado dessa última transformação, ilustrada pelo retângulo `contagens`, representa o RDD produzido na linha 9 da figura 3.3. De fato, transformar uma coleção de pares palavra/unidade em pares palavra/contagem pode demandar uma comunicação remota de contagens intermediárias. Dessa forma, a operação *c* não pode fazer parte do Estágio I e, consequentemente, marca o início do Estágio II. Em outras palavras, a operação *(c)* marca um *ponto de adaptação* da aplicação.

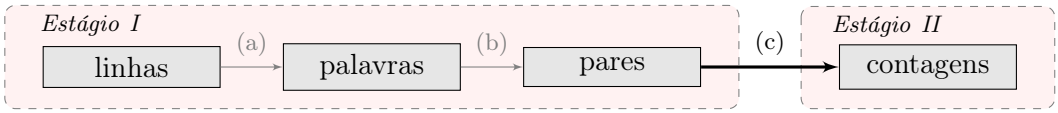


Figura 3.4: *Conta-palavras* em estágios. As transformações *a* e *b* representam mapeamentos e portanto geram dependências *estreita*. A transformação *c* representa uma redução global e portanto gera uma dependência *ampla*

Um estágio de execução em Spark pode ser descrito em termos de suas funções, dados de entrada, dados de saída, leitura de *shuffle* e escrita de *shuffle* (figura 3.5). Os dados de entrada podem representar dados provenientes de sistemas de armazenamento externo, como o HDFS Shvachko et al. [2010], ou mesmo RDDs persistidos em memória e/ou disco. Os dados de saída podem representar resultados de ações devem ser devolvidos ao driver ou escritos de volta em algum armazenamento externo. A escrita de *shuffle* representa os dados que as tarefas devem prover para o estágio posterior, armazenamento é feito nos discos locais dos nós de processamento participantes no processo. Finalmente, a leitura de *shuffle* representa os dados que cada tarefa deve buscar do estágio anterior para satisfazer as dependências globais por chave. Sendo assim, a presença ou ausência de tais dimensões especializam categorias de estágios:

- *ShuffleMap*: todo estágio que recebe como entrada dados (leitura de arquivos ou RDDs persistidos) e/ou *shuffle* e produz como saída uma **nova etapa de *shuffle***.

$$ShuffleMap = (DATAIN \vee SHIN) \wedge SHOUT$$

- *Result*: todo estágio que recebe como entrada dados (leitura de arquivos ou RDDs persistidos) e/ou *shuffle* e produz como saída o **resultado de uma ação** (escrita em arquivos ou conteúdo para o *driver*, como cardinalidade ou subconjunto de um RDD).

$$Result = (DATAIN \vee SHIN) \wedge DATAOUT$$

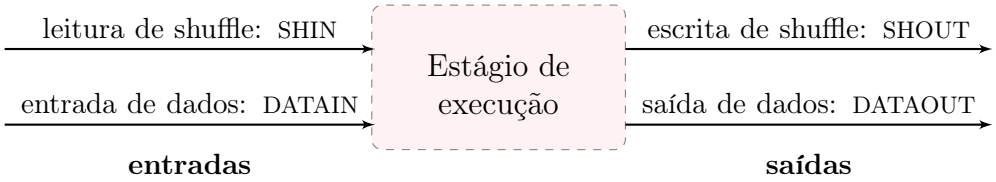


Figura 3.5: Formato de um estágio de execução

Como os estágios *ShuffleMap* terminam com uma escrita de *shuffle*, os dados produzidos devem ser particionados e armazenados temporariamente no disco local, para que todos os executores possam trocar registros remotamente. Esse processo envolve, na maioria dos casos, agregações e ordenações sobre o conjunto local de itens. Sendo assim, quanto maior a carga atribuída a cada tarefa/partição, maior será a quantidade de itens que precisam ser processados antes da escrita e consequentemente, maiores serão as chances de que o conjunto total de itens não caiba em memória. Nos casos em que isso acontece, ou seja, a memória disponível para uma tarefa é insuficiente para comportar todos os itens que precisam ser processados ao mesmo tempo, o sistema utiliza de mecanismos de ordenação externa para realizar a escrita do *shuffle* em várias rodadas. Cada rodada implica em um descarregamento de dados temporários em disco seguido de uma leitura para agregação, fato que impacta diretamente o desempenho de qualquer aplicação. A uma rodada desse processo damos o nome de *spill*, que por sua natureza é capaz de ser um fator determinante no desempenho das aplicações. Naturalmente, todo *spill* é indesejável e na maioria dos casos ele pode ser evitado ao se dividir mais a carga de cada tarefa, isto é, aumentar o número de partições do RDD. Esse fenômeno não acontece em estágios *Result*, pois nesse caso o *shuffle* não se aplica.

Os estágios identificados pelos DAGs são encaminhados para execução no momento da submissão de uma aplicação, na forma de *tarefas*. Cada tarefa é responsável por computar uma *partição* do RDD final no estágio corrente. Dessa forma, a cada tarefa temos associado um (e único) estágio de execução e, consequentemente, um conjunto de tarefas que executam o mesmo código referente a um subgrafo do DAG caracteriza um estágio.

O mestre da aplicação (programa *driver*) fica responsável por requisitar recursos no ambiente (seção 3.1), escalonar tarefas dos estágios e coordenar as dependências entre os mesmos. Em especial, alocação de recursos nesse contexto significa requisitar e alocar nós *executores* em prol da aplicação. A cada executor é associada uma quantidade de núcleos virtuais de processamento (*cores*) e memória, isto é, recursos que serão compartilhados para execução de tarefas dos estágios. Em geral, cada executor que possui mais de um *core* executa de diversas tarefas em paralelo, garantindo assim paralelismo local (em cada executor) e global (entre executores).

Variáveis para ajuste de desempenho

Uma forma de categorizar os ajustes possíveis para um programa é considerar sua área de impacto. Assim, um *ajuste global* influencia toda a execução e considera a aplicação como um todo. Em contrapartida, um *ajuste local* tem efeito em uma área específica do DAG de execução.

Spark permite ajustes globais através de configurações estáticas, escolhidas antes do início da computação propriamente dita, definidas na forma de pares chave/valor. Nesses casos, a chave identifica unicamente um parâmetro de configuração e o valor representa essa configuração instanciada. Como exemplo, o serializador utilizado para transferir os itens dos RDDs pela rede é selecionado através da configuração `spark.serializer` e uma possível instanciação para essa entrada é `JavaSerializer`, isto é, o serializador padrão da JVM.

Por outro lado, fazer um ajuste local em Spark significa alterar algumas propriedades estruturais dos RDDs da aplicação, ou seja, aplicar mudanças que impactam especificamente uma região do DAG de execução. As propriedades dos RDDs consideradas no contexto deste trabalho são o *número de partições* e o *particionador*. Como consequência do modelo de aplicação, há sempre a oportunidade de alterar essas propriedades para RDDs resultantes de dependência(s) do tipo *ampla*. Por simplicidade, não consideramos ajustes na topologia dos DAGs de execução. Assim, os ajustes locais impactam o particionamento dos RDDs e são condicionados a um único DAG que descreve a aplicação.

3.3 Dimensões de diagnóstico

Nesta seção, propomos as dimensões de diagnóstico que guiam nossa avaliação experimental. As discussões subsequentes serão sistematizadas de acordo com essas dimensões, que procuram cobrir os gargalos de desempenho conhecidos e que se relacionam

diretamente com os desafios listados. Além disso, as dimensões proveem uma terminologia comum para apresentar problemas e soluções.

3.3.1 Formato de dados

Em geral, formatos de armazenamento de dados são bem diferentes da forma com que os mesmos são manipulados por um programa. Estruturas de dados comuns carregam informações adicionais de semântica, que complementam e facilitam o acesso à informação original. Tipos primitivos e/ou compostos, arranjos contíguos e ponteiros são formatos comuns utilizados para o processamento em memória. Além disso, linguagens orientadas a objetos incorporam uma camada de abstração que contribui para produtividade e modularidade, por isso são predominantemente utilizadas como ferramentas de desenvolvimento de plataformas de processamento paralelo, como Hadoop e Spark. Entretanto, essa flexibilidade vem com um custo adicional em utilização de memória.

Em especial, esses ambientes executam sobre uma máquina virtual (*Java Virtual Machine*, ou JVM) que é conhecida por sua sensibilidade quanto ao formato de dados e padrões de acesso aos mesmos [Nguyen et al., 2015]. Por isso, é importante levar em conta a formatação dos dados utilizada pelas implementações. Afinal de contas, aplicações podem sempre balancear o compromisso entre abstrações de alto nível por clareza (por exemplo, objetos e estruturas complexas) e estruturas de baixo nível por desempenho em grão fino (por exemplo, arranjos simples e tipos primitivos).

3.3.2 Localidade de tarefas

Sistemas de processamento baseados em paralelismo de dados dividem trabalho ao disparar diversas tarefas que executam o mesmo código (estágio) em diferentes partes do dado. Com esse funcionamento, escalar a computação em relação ao tamanho da entrada é direto, já que o particionamento dos dados dita o nível de paralelismo. Entretanto, considerando-se que aplicações processam giga- ou mesmo terabytes de dados, a qualquer momento, centenas/milhares de tarefas podem estar prontas para execução. É papel do escalonador receber essa carga de trabalho, organizá-la através de um plano de execução e decidir qual tarefa deve ser alocada em qual recurso.

O escalonador deve ainda lidar com o compromisso entre vazão de tarefas completas e qualidade das alocações. Em muitas aplicações, tarefas podem ter custos computacionais diferentes, ou processar diferentes quantidades de dados. Isso acontece por dois motivos: *(i)* possibilidade que usuários têm de injetar código genérico nos operadores e *(ii)* o desconhecimento de antemão da distribuição dos dados. Por isso,

a co-alocação de tarefas heterogêneas tem o potencial de criar gargalos inesperados e impactar no desempenho da aplicação. Por exemplo, atribuir muitas tarefas computacionalmente caras a um mesmo subconjunto de recursos poderia aumentar a chance de saturar alguns nós de processamento e ao mesmo tempo causar subutilização de outros.

3.3.3 Paralelismo adequado

Alcançar o grau correto de paralelismo não significa apenas ajustar as aplicações para melhor desempenho, mas fazê-lo utilizando apenas o suficiente de recursos em um dado momento da execução. Em sistemas de processamento paralelo, a entidade responsável por lidar com o paralelismo da computação é o *particionador*, que organiza os itens dos dados baseando-se em um *número de partições*. É desejável que o particionador seja capaz de se adequar ao comportamento de cada execução no que se refere ao nível de paralelismo. Esse aspecto se torna ainda mais crítico em *frameworks* da nova geração, onde um programa é composto de diversos estágios encadeados por dependências *amplas*. Cada estágio, portanto, pode ter um nível diferente de paralelismo e a oportunidade para esse ajuste aparece durante cada processo de *shuffle* enquanto os dados estão sendo reorganizados para o próximo estágio. Assim, o particionamento ideal deve ser tratado individualmente para cara estágio, o que pode implicar em um comportamento elástico.

3.3.4 Balanceamento de carga

Além da quantidade de partições de uma computação, é preciso se preocupar com o custo das tarefas associadas a cada partição, visto que algoritmos podem refletir cargas heterogêneas. Como visto neste capítulo, o modelo de execução do ambiente Spark cria barreiras implícitas no início de cada estágio, necessárias para satisfazer as dependências de dados antes de prosseguir com qualquer computação. Assumindo, sem perda de generalidade, que aplicações executam estágios sequencialmente (estágios não dependentes podem executar concorrentemente), todo desbalanceamento nas tarefas de um estágio pode levar à ociosidade de recursos, que por sua vez pode criar consideráveis gargalos em desempenho. Tal desbalanceamento pode ser inerente ao algoritmo sendo executado ou devido a um particionamento ruim. Apenas com uma análise detalhada de uma execução podemos determinar a estratégia para lidar com cada caso específico.

O modelo de execução e as dimensões de diagnóstico apresentados neste capitulo servem de base para a definição da carga de trabalho considerada e sua caracterização

do ponto de vista de desempenho em processamento de dados massivos. Esses dois aspectos são detalhados e discutidos a seguir, nos capítulos 4 e 5.

Capítulo 4

Algoritmos e padrões de comunicação

Um fluxo de trabalho comum em processamento de dados massivos envolve etapas de limpeza, exploração, caracterização e modelagem de grandes volumes de dados. Realizar tais tarefas em sistemas paralelos e distribuídos tem a vantagem de aumentar o poder de escalabilidade e aplicabilidade dos algoritmos que compõem essas etapas. Mais importante, quando fixamos um modelo de execução e um conjunto de procedimentos padrão, começamos a perceber comportamentos comuns nas aplicações. Tomando como base o modelo de execução representado através de DAGs, podemos observar que muitos algoritmos possuem uma estrutura de execução semelhante, tanto entre aplicações quanto dentro de uma mesma aplicação.

Por isso, argumentamos que tais semelhanças podem ser úteis para categorizar aplicações em grupos menores. O objetivo é que sejamos capazes de entender o padrão de comunicação dessas aplicações e utilizar essa informação para construir estratégias de otimização de desempenho mais fundamentadas e reaproveitáveis.

A definição e a escolha das categorias foram baseadas em dois aspectos que podem ser usados para determinar o comportamento de uma aplicação: iteratividade e regularidade. O primeiro se refere à estrutura do programa, indicando se o procedimento é iterativo ou não. De fato, iteratividade é uma característica marcante de muitos algoritmos de mineração de dados e aprendizado de máquina comumente usados para extrair conhecimento de bases de dados. O segundo representa diretamente o relacionamento entre programa e dados de entrada, que juntos formam o padrão de comunicação da aplicação. Dessa forma, caracterizamos como regular uma aplicação que repete um padrão de comunicação várias vezes, processando o mesmo volume de dados em cada iteração.

A iteratividade e a regularidade implicam em três categorias de aplicações: *(i)* não-iterativas; *(ii)* iterativas regulares; e *(iii)* iterativas irregulares. A seguir detalhamos cada categoria e descrevemos uma aplicação representativa para cada um, que servirá como carga de trabalho para os testes ao longo da dissertação.

4.1 Algoritmos não-iterativos

Uma aplicação é classificada como não-iterativa do ponto de vista de sua representação em DAG se suas principais subestruturas de execução não se repetem durante uma mesma execução. Assim, se enquadram nesse grupo aplicações que representam sequências muito bem definidas de passos, com a ausência de dependências cíclicas. A aplicação canônica para justificar *big data*, o WordCount, exibe um comportamento como esse. A base de dados é lida, transformada, reduzida e re-escrita no sistema de arquivos distribuído. Nesse sentido, o DAG de execução tem uma estrutura bem direta e nenhum procedimento é repetido ao longo do tempo.

Do ponto de vista de oportunidades, otimizar aplicações não-iterativas requer ou instrumentações sofisticadas para tomadas de decisão *online* ou uma observação do histórico de execução caso o seu propósito seja recorrente. Neste trabalho exploramos e validamos a segunda opção. A seguir descrevemos o algoritmo Twidd, que servirá de modelo representativo para a categoria de aplicações não-iterativas. Outros algoritmos que exibem a mesma característica do Twidd e portanto se enquadram nessa categoria de aplicações incluem: consultas que calculam relatórios ou realizam análises detalhadas de um conjunto de dados, isto é, aplicações ETL em geral (**E**xtract, **T**ransform,**L**oad); alguns algoritmos clássicos de *big data* como *Nutch Indexing*, *Sort* e *WordCount*.

Exemplo: Twidd

O Twidd é uma implementação do FPGrowth [Han et al., 2000] sobre a abstração de RDDs. Ele resolve o problema de mineração de padrões frequentes, cujo objetivo é: dado um conjunto de transações, cada qual composta por conjuntos de itens, e um limiar de suporte *minsupp*, encontrar todos os subconjuntos de itens (*itemsets*) que ocorram em mais de *minsupp* transações. Uma característica marcante é o fato de ser um algoritmo não-iterativo do ponto de vista do DAG do programa (e aqui está o nosso interesse), apesar de ser considerado iterativo do ponto individual de algumas UDFs da implementação.

A maioria das abordagens iniciam com a construção de uma tabela de frequência global dos 1-itemsets (isto é, subconjuntos de itens de cardinalidade 1, que representa a frequência de cada item individualmente). A descrição a seguir assume que essa tabela foi calculada e está replicada em todo executor do ambiente Spark. No caso do FPGrowth, a computação continua com a construção de árvores de prefixos locais utilizando itens conhecidamente frequentes de acordo com a tabela.

Uma visão geral do Twidd pode ser observada na figura 4.1. O algoritmo inicia lendo as transações do sistema de arquivos. O resultado é um RDD de *transações*, particionado entre os executores da aplicação. Os subconjuntos de transações, organizados em partições, servem de entrada para a construção de árvores de prefixo locais, que são obtidas através da adição de itens e suas respectivas frequências absolutas (informação contida na tabela de frequências). O resultado são árvores locais por partição (*FPTrees locais*), construídas sobre os subconjuntos de transações (transformação a) e que mantêm itens mais frequentes próximos às respectivas raízes.

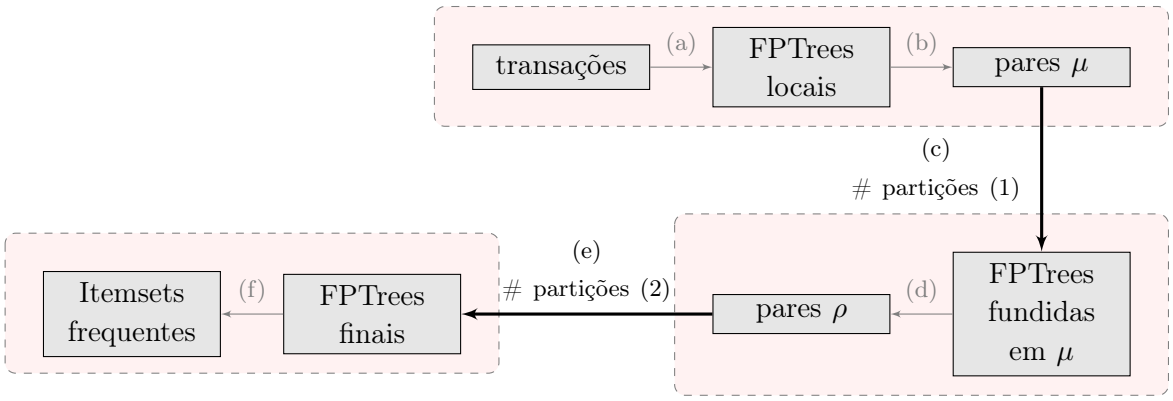


Figura 4.1: Visão geral do algoritmo *Twidd*

O próximo passo consiste de extrair μ-árvores das árvores locais a partir de prefixos de tamanho μ (transformação b). Esse procedimento serve como um balanceamento de carga intermediário característico do algoritmo, percorrendo as árvores locais até o nível μ e gerando prefixos e subárvores (pares) para o primeiro *shuffle* da aplicação. A ideia é mitigar desbalanceamento ao replicar nós das árvores em mais de uma partição. Para isso, mapeamos cada elemento do RDD de árvores locais em potencialmente diversas subárvores (pares obtidos por uma operação um-para-muitos). O resultado é, portanto, um novo RDD de subárvores chaveadas pelos respectivos prefixos extraídos de tamanho μ. A etapa de *shuffle* ocorre (transformação c) e subárvores com prefixos comuns são fundidas em um novo RDD.

Os próximos passos representam as projeções do algoritmo FPGrowth. Em especial, árvores fundidas no passo anterior são condicionadas até o nível ρ (transformação d), resultando em um novo RDD de pares (*pares* ρ), de forma similar ao que foi feito na transformação c. Se trata de uma pré-projeção e conclui a estratégia de balancemento do Twidd. Então, um segundo *shuffle* garante que subárvores condicionadas sobre o mesmo prefixo serão fundidas (transformação e). O resultado é um RDD de FPTrees finais que então são projetadas independentemente. Por fim, os itemsets frequentes encontrados são escritos no HDFS.

Avaliamos o Twidd em termos das dimensões *Formato de Dados*, *Localidade de Tarefas* e *Balanceamento de Carga*, devido à sua implementação baseada em estruturas complexas (árvores) que têm o potencial de expor diferentes comportamentos quanto à coleta de lixo (GC) e cargas de trabalho heterogêneas entre tarefas.

4.2 Algoritmos iterativos e regulares

Apesar do DAG de execução, por definição, não conter ciclos ainda é possível representar aplicações iterativas de qualquer natureza. A estratégia atualmente utilizada é justamente repetir e encadear as subestruturas que compõem o programa. Dessa forma o escalonador do ambiente pode, a princípio, se torna agnóstico do padrão de comunicação das aplicações. Se por um lado isso simplifica o modelo de programação, por outro dificulta a associação de partes do programa que poderiam se beneficiar mutuamente. Como veremos a seguir, experiências com iterações iniciais podem ser úteis para reconfigurar iterações posteriores, desde que o ambiente de execução esteja ciente do comportamento iterativo da aplicação. Aplicações iterativas podem ainda ser classificadas em relação à sua regularidade (ou ausência dela). Nesta seção tratamos de aplicações iterativas regulares, ou seja, que realizam o mesmo padrão de comunicação e volume de dados em cada iteração.

A seguir descrevemos o PageRank, a aplicação modelo que utilizamos para representar algoritmos iterativos e regulares. Outros algoritmos que possuem esse mesmo padrão de comunicação regular e por isso se enquadram nessa categoria de aplicações incluem: *KMeans* e *Collaborative Filtering* com **A**lternating **L**east **S**quares (ALS).

Exemplo: PageRank

O PageRank [Page et al., 1999] é um algoritmo analítico que computa a importância relativa de nós em uma rede. Ele o faz atribuindo rankings iniciais para todo nó e iterativamente atualizando esses valores levando em conta cada vizinhança em questão.

Inicialmente, o ranking de todo nó é 1. A cada iteração, todo nó divide seu próprio ranking entre seus vizinhos. Então, um nó com ranking r e n vizinhos compartilha $\frac{r}{n}$ de seu próprio valor com cada vizinho. Em seguida cada nó recebe as contribuições de seus vizinhos e somando esses valores constrói o seu novo ranking. O processo continua até que um dado número de iterações seja atingido ou até que os valores convirjam de acordo com um erro tolerado. O PageRank é iterativo pois se trata de um algoritmo de convergência. Além disso ele é regular pois a cada iteração, o mesmo conjunto de mensagens é trocado entre os procedimentos.

A figura 4.2 ilustra o funcionamento do algoritmo. Os links são carregados do sistema de arquivos e agregados através da representação de listas de adjacências (*links*). O RDD resultante é mantido em memória (*cache*) para futuro reuso. Então, atribuímos o ranking inicial (inicialmente 1) para os links (transformação a). Uma junção é realizada entre os rankings correntes e a adjacência dos nós, o que permite o cálculo das contribuições de rankings na rede (transformação b). Atualizamos então os rankings de todos os nós ao somar as contribuições individuais em cada nó. O resultado é um novo RDD de rankings, que realimenta o ciclo.

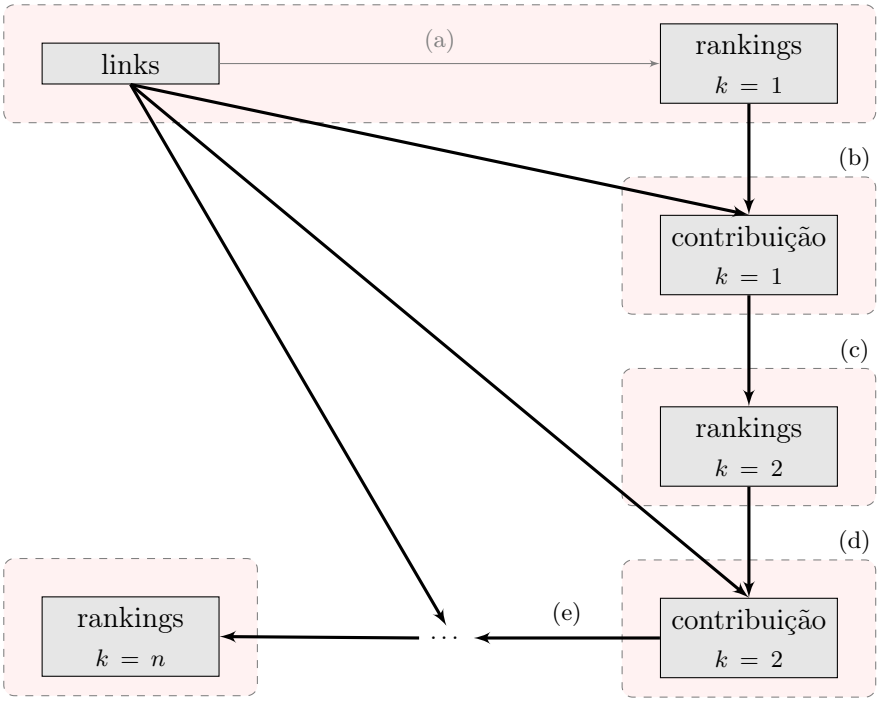


Figura 4.2: Visão geral do algoritmo *PageRank*

A cada iteração o PageRank realiza a mesma computação em todos os nós, configurando um comportamento regular. Nós o avaliamos em termos da dimensão *Paralelismo Adequado*, pelo seu potencial de esclarecer oportunidades de otimização para

algoritmos com comportamentos regulares e iterativos. De fato, cada iteração do algoritmo produz o mesmo número de mensagens, pois essa quantidade é determinada pela adjacência dos vértices do grafo. Essa repetição de padrões dentro de uma mesma execução sugere que o conhecimento obtido com etapas anteriores pode ajudar a otimizar iterações posteriores.

4.3 Algoritmos iterativos e irregulares

A última categoria se refere aos algoritmos que, apesar de iterativos, exibem um comportamento variável do ponto de vista dos padrões de comunicação. Dessa forma, os procedimentos realizados a cada iteração continuam sendo os mesmos, porém os dados de entrada/saída para cada uma delas variam em natureza e/ou dimensão. Classificamos esse comportamento como sendo irregular. Oportunidades para otimização de programas dessa natureza podem se beneficiar da informação de que o mesmo código está sendo re-executado várias vezes, mas deve considerar que, dependendo da evolução do algoritmo, o custo associado aos dados pode influenciar e até mesmo dominar o desempenho como um todo.

A seguir descrevemos o Eclat, o algoritmo modelo que utilizamos para representar essa classe de aplicações. Outros algoritmos que apresentam esse mesmo comportamento iterativo irregular e portanto se enquadram nessa categoria de aplicações incluem: algoritmos construídos sobre o modelo Pregel [Malewicz et al., 2010] como Contagem de triângulos e Componentes Conectados; Regressão Logística, entre outros.

Exemplo: Eclat

O Eclat, como o Twidd, é também um algoritmo para mineração de padrões frequentes, que trabalha sobre uma representação vertical da base de dados. Assim, ao invés de um conjunto de itens para cada transação, temos conjuntos de identificadores de transações para cada item. Isso permite uma contagem e geração eficiente de candidatos através da intersecção das listas invertidas de transações (*t-lists*). O Eclat é iterativo pela sua característica hierárquica durante a mineração e irregular pela combinatória envolvida na distribuição dos itemsets frequentes em relação aos seus tamanhos.

Por simplicidade, como no Twidd, a seguinte descrição da implementação do Eclat sobre Spark assume que a tabela de frequências globais para 1-itemsets está disponível para consulta em todo *cluster*. Nós adotamos uma estratégia de contagem local [Veloso et al., 2004] para implementar o Eclat sobre a abstração de RDDs. Ela combina a

geração eficiente de candidatos do Eclat (intersecção de t-lists) para contagem local com uma etapa de *shuffle* para agregação global.

A figura 4.3 apresenta a visão geral da implementação. O procedimento começa com a leitura das transações como um RDD de listas de itens. Em seguida, realizamos a verticalização da base através da construção das listas invertidas, como demandado pelo Eclat (transformação a). Entretanto, cada partição do RDD é verticalizado independentemente, o que evita que existam t-lists demasiadamente grandes. Por isso, nossa implementação mitiga uma limitação conhecida do Eclat ao se aproveitar da representação particionada de um RDD.

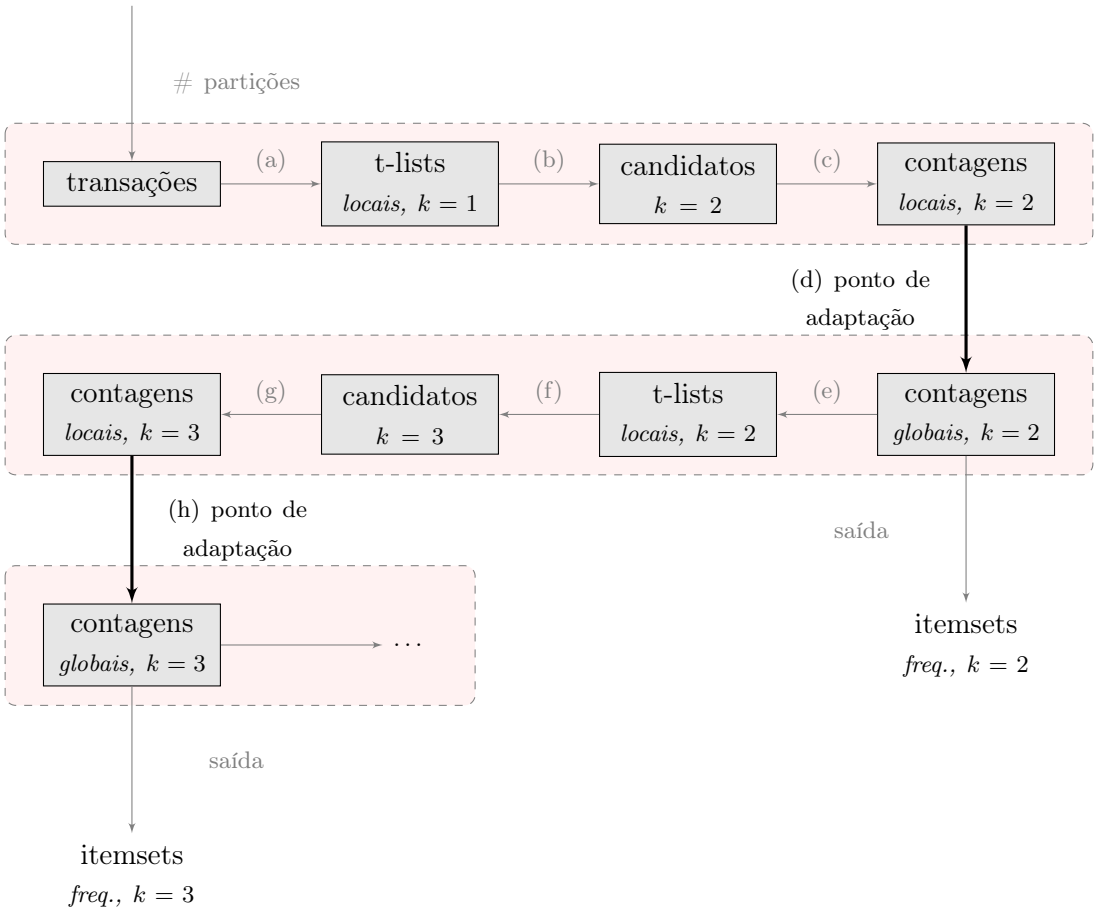


Figura 4.3: Visão geral do algoritmo *Eclat*

O restante dos itemsets frequentes são minerados de forma hierárquica e iterativa. Assim, geramos $(k+1)$-itemsets candidatos ao realizar a intersecção das t-lists correntes (transformação b). Contagens locais são extraídas desses novos candidatos (transformação c) e agregadas globalmente (transformação d). Esse resultado tem dois fins: adicionar os $(k+1)$-itemsets frequentes como saída do algoritmo e filtrar candi-

datos infrequentes para construir novas t-lists para a próxima iteração (transformação *e*). Esse ciclo continua até que nenhum novo itemset frequente seja encontrado.

Nós avaliamos o Eclat em termos da dimensão *Formato de Dados* devido à sua implementação baseada em estruturas de dados simples (arranjos e tipos primitivos, apenas). Além disso, o Eclat possui um compromisso entre o número de bases locais (isto é, a quantidade de partições verticalizadas independentemente) e o grau de replicação das t-lists, que, juntamente com sua natureza hierárquica e combinatória, exibem um potencial para um estudo da dimensão de *Paralelismo Adequado*.

Capítulo 5

Caracterização do desempenho

A complexidade e a composição de níveis de abstração nos sistemas paralelos atuais dificultam o estudo de um sistema do ponto de vista puramente analítico. Por esse motivo, adotamos uma avaliação experimental preliminar com o intuito de caracterizar fontes de ineficiência comuns do ponto de vista das dimensões de diagnóstico apresentadas. O resultado deste capítulo é um conjunto de constatações e oportunidades para melhorar o desempenho das categorias de aplicações estabelecidas como carga de trabalho. A seção 5.1 fornece a especificação dos recursos computacionais e as bases de dados utilizados nos experimentos. A seção 5.2 detalha o processo de coleta e interpretação das métricas disponibilizadas pelo sistema que serão utilizados na extração de todo resultado apresentado. Finalmente, na seção 5.3, os casos de teste são descritos e seus resultados analisados de acordo com as correspondências entre dimensões de diagnóstico e aplicações determinadas no capítulo 4.

5.1 Configuração do ambiente

O ambiente utilizado na avaliação experimental é composto de 9 máquinas, cada um contendo um processador quad-core Intel Xeon X3440 com *hyperthreading* e 8 MB de cache, 16 GB de memória RAM, um disco SATA de 1 TB e 7200 RPM, executando um sistema operacional Linux 3.2.0 de 64 bits. Os nós de processamento estão conectados com uma rede Gigabit Ethernet. O *cluster* foi configurado com Spark v1.5.1 e Hadoop/HDFS v2.5.0. Duas bases de dados reais foram utilizadas nos experimentos: um conjunto de posts do Twitter, contendo tweets coletados usando a API da plataforma; e um grafo da rede social Google+ [Magno et al., 2012] representando usuários (vértices) e amizades (arestas). Todas as bases de dados foram carregadas no HDFS com um fator de replicação igual a 2. A tabela 5.1 resume as configuraçoes de algoritmos e

bases de dados.

Nós conduzimos uma avaliação experimental dos algoritmos descritos no capítulo 4 sobre as dimensões de diagnóstico introduzidas na seção 3.3.

Aplicação	***Base de dados***	***Tamanho***	***Blocos do HDFS***	***Detalhes***
Twidd/Eclat	twitter	7,9 GB	63	# transações: 233 mi # itens distintos: 64 mi
PageRank	gplus	9,3 GB	75	# vértices: 35 mi # arestas: 575 mi

Tabela 5.1: Algoritmos e bases de dados

5.2 Execução e coleta de resultados

A figura 5.1 apresenta o fluxo de coleta e extração das estatísticas utilizadas na análise de resultados. Seguindo a numeração indicada na figura, a aplicação de teste é submetida (*passo 1*) e produz o seu resultado tradicional como saída no HDFS (*passo 2*) e também registra ao longo de sua execução os eventos no repositório de logs (*passo 3*) em formato padrão do Spark (JSON, assim como nas amostras das figuras 5.2 e 5.3). Essa geração é feita pelo mestre da aplicação (também conhecido como programa *driver* no linguajar do ambiente), por isso essa saída pode ser realizada no sistema de arquivos local. A última etapa consiste da análise estática dos logs produzidos, guiados pelas dimensões de diagnóstico e características de cada grupo de aplicações (*passo 4*). O processo de análise dos logs é feito utilizando scripts comuns em linguagem python, por dois motivos: *(i)* pela facilidade de manipulação de dados no formato JSON; e *(ii)* pela dimensão reduzida do tamanho dos logs, viabilizando uma análise sequencial simples. O nosso objetivo com essa etapa é explorar mais a fundo aspectos não-óbvios que caracterizem as principais fontes de ineficiência de aplicações típicas, o resultado esperado é que o conjunto de lições possa auxiliar no implantação de estratégias automatizadas para ajuste de desempenho.

Por ser uma plataforma integrada, o Spark conta com funcionalidades de monitoração e instrumentação bem completas. A granularidade de coleta de estatísticas no sistema é no nível de tarefa. Desta forma, toda tarefa contabiliza, ao longo de sua execução, uma série de métricas de desempenho e contexto referentes à computação da partição em questão. Essa informação é empacotada, serializada e despachada para o programa principal de uma aplicação (*driver*) ao término de cada tarefa. As métricas (por tarefa) coletadas pelo sistema e relevantes para nossa discussão são descritas a

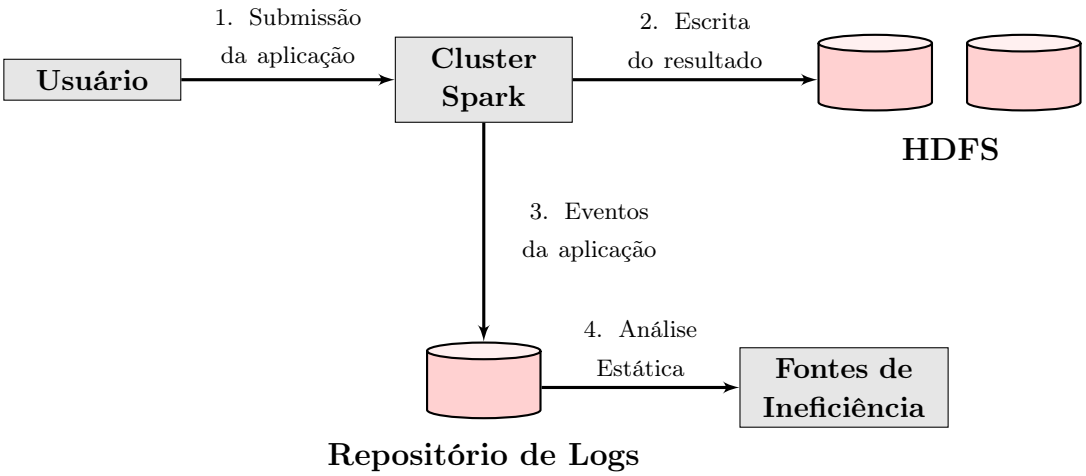


Figura 5.1: Fluxo de coleta e análise para caracterização das aplicações

seguir. Os nomes entre colchetes indicam a hierarquia da métrica no formato de saída, como amostrado nas figuras 5.2 e 5.3.

- **Início** `[ Task Info, Launch Time ]`:
  O *timestamp* de disparo da tarefa pelo escalonador do ambiente.
- **Término** `[ Task Info, Finish Time ]`:
  O *timestamp* no qual a tarefa completou, ou seja, quando o mestre foi notificado e receber o seu resultado.
- **Tempo de execução** `[ Task Metrics, Executor Run Time ]`:
  Tempo de execução útil da tarefa ignorando gastos adicionais como empacotamento e desempacotamento e distribuição do executável entre os executores.
- **Tempo de coleta de lixo** `[ Task Metrics, JVM GC Time ]`:
  Tempo que o executor responsável por executar a tarefa gastou com coleta de lixo.
- **Métricas da entrada** `[ Task Metrics, Input Metrics ]`:
  Se referem aos dados de entrada da tarefa obtidos através de armazenamento externo (arquivos) ou RDDs existentes na execução corrente.

  Bytes lidos `[ Bytes Read ]`: Dimensão dos dados em bytes.

  Registros lidos `[ Records Read ]`: Número de registros e, no caso de RDDs, o número de itens da partição.
- **Métricas de shuffle/leitura** `[ Task Metrics, Shuffle Read Metrics ]`:
  Se referem aos dados de entrada obtidos através da leitura do *shuffle* de estágios de dependência, isto é, estágios ancestrais do tipo *ShuffleMap*.

Bytes lidos remotamente `[ Remote Bytes Read ]`: Parte dos dados lidos pela rede, em bytes.

Registros lidos `[ Total Records Read ]`: Número de registros.

- **Métricas de shuffle/escrita** `[ Task Metrics, Shuffle Write Metrics ]`: Se referem aos dados intermediários particionados e escritos por um estágio do tipo *ShuffleMap*. Esse *shuffle* pode ser consumido por um estágio *ShuffleMap* ou *Result*, dependendo se a aplicação irá ou não retornar resultado para o programa de usuário no driver, no caso da execução de uma ação.

  Bytes escritos `[ Shuffle Bytes Written ]`: Tamanho da escrita do *shuffle*, em bytes.

  Shuffle Records Written `[ Shuffle Records Written ]`: Número de registros escritos no *shuffle*.

O sistema disponibiliza essas métricas em um formato conhecido e que facilita uma posterior análise: JSON (*JavaScript Object Notation*). A persistência dos logs de execução pode ser ativada pelo usuário através da configuração `spark.eventLog.enabled`. A partir dos logs de execução que contêm as métricas coletadas pelo sistema, consideramos as seguintes grandezas para caracterizar o desempenho das aplicações:

1. **Tempo de execução** (TE): se refere ao tempo de resposta total de uma execução, podendo se referir ao tempo de uma aplicação ou tempos individuais de cada tarefa. Essa métrica nos permite avaliar a aplicação do ponto de vista prático, isto é, sua eficiência para qualquer observador externo.

2. **Overhead com coleta de lixo** ($\%GC$): porcentagem do tempo no qual o executor responsável pela execução da tarefa gastou com coleta de lixo (***G**arbage **C**ollection*) em relação ao tempo de execução (TE). Essa métrica nos permite avaliar uma das principais fontes de ineficiência em processamento massivo de dados: gerenciamento de memória.

3. **Tamanho da entrada** (TIN): se refere ao tamanho da entrada de um estágio de execução. Pode significar o tamanho de um arquivo sendo lido ou o tamanho de um RDD servindo como dependência para a aplicação.

4. **Tamanho do shuffle** (TSH): se refere à quantidade de dados em bytes lida por um estágio de *shuffle*. Quanto maior o *shuffle*, mais comunicação precisa acontecer entre os executores da aplicação.

As figuras 5.2 e 5.3 representam trechos do arquivo JSON de saída de uma execução. A primeira representa um evento de término de tarefa de um estágio *Result*. Sabemos disso pela entrada *"Task Type"* e também pelo fato de que as métricas referentes à escrita do *shuffle* não estão presentes. A segunda figura, por outro lado, representa um evento de término de tarefa de um estágio *ShuffleMap*: note a existência de métricas relacionadas à etapa de escrita do *shuffle*. Essa distinção é importante pois, além de refletir o modelo de execução do ambiente, indica as possibilidades de otimização que podem ser feitas em benefício da aplicação. Por exemplo, identificando que um determinado estágio é do tipo *ShuffleMap*, temos portanto a oportunidade de mudar o paralelismo nesse ponto.

5.3 Resultados e análise

Esta seção apresenta um conjunto de análises estáticas (isto é, realizadas após a execução de uma aplicação) com o objetivo de motivar a importância do ajuste de parâmetros para otimização do desempenho, estudar detalhadamente fontes de ineficiência não óbvias e destacar as oportunidades e/ou limitações emergentes das discussões. Para tanto, a seção 5.3.1 realiza um estudo de alguns parâmetros do ambiente frente a um conjunto de métricas, a seção 5.3.2 destaca a importância do particionamento de dados e alocação de tarefas, a seção 5.3.3 discute como o comportamento das aplicações pode ajudar na otimização de desempenho. Finalmente, a seção 5.3.4 discute como um mesmo gargalo em desempenho pode ter causas diversas e, mais importante, como um diagnóstico detalhado pode ajudar na mitigação desses problemas.

5.3.1 Projeto fatorial

Projetos fatoriais completos são utilizados para estimar os efeitos de fatores sobre um sistema do ponto de vista de uma variável de resposta (ou métrica de avaliação). Os fatores representam as dimensões de estudo e níveis definem as variações sobre essas dimensões.

Uma especialização desse tipo de projeto é o fatorial 2^k, no qual consideramos apenas dois níveis para cada um de k fatores, resultando em 2^k possíveis configurações de avaliação. Isso implica na execução de $r2^k$ experimentos, onde r é o número de replicações que serão utilizadas para derivar o intervalo de confiança das asserções. A saída do processo é um modelo para cada métrica que associa variações na métrica aos fatores considerados, suas interações e qualquer erro, isto é, porcentagem de variação que não foi atribuída a nenhum dos fatores estudados. Assim, se o efeito atribuído a

```
{
  "Event": "SparkListenerTaskEnd",
  "Stage ID": 0,
  "Stage Attempt ID": 0,
  "Task Type": "ResultTask",
  "Task End Reason": {
    "Reason": "Success"
  },
  "Task Info": {
    "Task ID": 9,
    "Index": 9,
    "Attempt": 0,
    "Launch Time": 1465441668345,
    "Executor ID": "driver",
    "Host": "localhost",
    "Locality": "PROCESS_LOCAL",
    "Speculative": false,
    "Getting Result Time": 0,
    "Finish Time": 1465441668430,
    "Failed": false,
    "Accumulables": [
    ]
  },
  "Task Metrics": {
    "Host Name": "localhost",
    "Executor Deserialize Time": 7,
    "Executor Run Time": 70,
    "Result Size": 2772,
    "JVM GC Time": 0,
    "Result Serialization Time": 0,
    "Memory Bytes Spilled": 0,
    "Disk Bytes Spilled": 0,
    "Input Metrics": {
      "Data Read Method": "Hadoop",
      "Bytes Read": 5986,
      "Records Read": 62
    },
    "Updated Blocks": [
      . . .
    ]
  }
}
```

Figura 5.2: Linha do log contendo um evento de término de tarefa e respectivas métricas. Essa entrada se refere a uma tarefa *Result*, como indicado no campo "Task Type"

```
{
   "Event":"SparkListenerTaskEnd",
   "Stage ID":4,
   "Stage Attempt ID":0,
   "Task Type":"ShuffleMapTask",
   "Task End Reason":{
      "Reason":"Success"
   },
   "Task Info":{
      "Task ID":378,
      "Index":124,
      "Attempt":0,
      "Launch Time":1465440465902,
      "Executor ID":"driver",
      "Host":"localhost",
      "Locality":"NODE_LOCAL",
      "Speculative":false,
      "Getting Result Time":0,
      "Finish Time":1465440465923,
      "Failed":false,
      "Accumulables":[

      ]
   },
   "Task Metrics":{
      "Host Name":"localhost",
      "Executor Deserialize Time":7,
      "Executor Run Time":5,
      "Result Size":1431,
      "JVM GC Time":0,
      "Result Serialization Time":0,
      "Memory Bytes Spilled":0,
      "Disk Bytes Spilled":0,
      "Shuffle Read Metrics":{
         "Remote Blocks Fetched":0,
         "Local Blocks Fetched":7,
         "Fetch Wait Time":0,
         "Remote Bytes Read":0,
         "Local Bytes Read":112,
         "Total Records Read":0
      },
      "Shuffle Write Metrics":{
         "Shuffle Bytes Written":0,
         "Shuffle Write Time":104170,
         "Shuffle Records Written":0
      }
   }
}
```


Figura 5.3: Linha do log contendo um evento de término de tarefa e respectivas métricas. Essa entrada se refere a uma tarefa *ShuffleMap*, como indicado no campo "Task Type"

um determinado fator for alto então temos um forte indício de que esse fator é o que tende a influenciar a métrica de avaliação.

Adotamos o projeto fatorial 2^k para estimar as porcentagens de variação de parâmetros chave em aplicações Spark sob diferentes circunstâncias. Avaliamos o Twidd e Eclat em diferentes tamanhos de *cluster* (5 e 9 máquinas). Os fatores foram escolhidos baseando-se em parâmetros comuns a qualquer aplicação (memória e *cores* dos executores) e no número de partições em diferentes etapas dos algoritmos, nos *pontos de adaptação* indicados nas figuras 4.1 e 4.3. Com o objetivo de simplificar as discussões, apresentamos apenas os resultados obtidos pelo grupo de modelos de melhor qualidade: projetos com 5 executores. As considerações sobre a qualidade e premissas dos projetos fatoriais estão reunidas no apêndice A.

Os fatores e níveis usados para configurar cada algoritmo estão resumidos na tabela 5.2, sendo que cada configuração foi replicada cinco vezes. Note como escolhemos dois níveis para cada um dos fatores, um baixo e um alto. Com isso, nosso propósito é observar como cada fator afeta o tempo de execução e entender como eles influenciam na porcentagem de tempo gasta com coleta de lixo (métrica %GC).

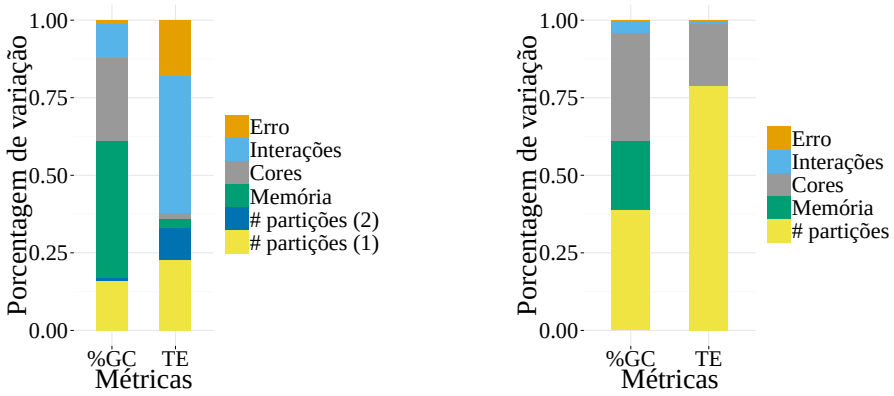
Twidd	*Mem. Exec.*	*Cores Exec.*	*# Partições (1)*	*# Partições (2)*
	7 GB	4	257	257
	14 GB	8	1021	1021
Eclat	*Mem. Exec.*	*Cores Exec.*	*# Partições*	
	7 GB	4	257	
	14 GB	8	1021	

Tabela 5.2: Parâmetros do projeto fatorial $2^k r$. Os projetos consideram cinco replicações para cada configuração, isto é, $r = 5$.

A figura 5.4 ilustra os resultados obtidos para o Twidd. Observe que o erro para o modelo de %GC é pequeno (1%) e por isso o tempo gasto em coleta de lixo é bem definido para os fatores considerados, isto é, muito da variação de %GC pôde ser explicada pelos fatores de interesse. Pelo menos 80% da variação de %GC é explicada por memória e quantidade de *cores* isoladamente. Esse comportamento é esperado: mais memória e menos *cores* implicam em um maior espaço de memória heap para ser varrida e coletada com menos poder computacional.

Diferentemente do modelo de %GC, o modelo do tempo de execução (**TE**) está sujeito a um erro maior (26,2%), indicando que as variações não estão sendo bem explicadas pelos fatores do estudo. O Twidd faz uso de estruturas complexas que inflacionam o tempo gasto em coleta de lixo [Nguyen et al., 2015]. De fato, coleta de lixo por si só é conhecida como um dos principais fatores responsáveis pela imprevi-

sibilidade no desempenho de aplicações paralelas nos ambientes considerados [Maas et al., 2015]. Além disso, o modelo é melhor explicado pelas interações entre fatores (53,7%), ao invés de fatores isolados. Uma análise mais detalhada (não presente na figura) revela que dentre as possíveis interações, a mais impactante foi a entre memória e quantidade de *cores*. Isso indica que a implementação possui uma proporção ideal de memória por *core* que leva a melhores resultados. Tais interações podem depender de elementos como o comportamento das funções injetadas pelo usuário nos operadores do ambiente (UDFs), como elas utilizam memória, e o tipo de computação que realizam. Dependendo da aplicação e sua carga de trabalho, as demandas por processamento e memória podem ser bastante discrepantes. Nesses casos, a abundância de um recurso não crítico pode ter inclusive um efeito negativo no desempenho da aplicação. Por exemplo, se o recurso dominante de uma aplicação for memória, aumentar o número de *cores* significa diminuir a quantidade de memória alocada por core da aplicação e causar uma diminuição generalizada da eficiência do trabalho realizado por cada um desses *cores*. Em particular, esse foi o cenário encontrado durante a avaliação do Twidd, dado a complexidade de suas estruturas de dados e o impacto negativo que isso tem na coleta de lixo na memória. Por esses motivos, lidar com a alocação de múltiplos recursos e suas implicações no desempenho são desafios cada vez mais evidentes em sistemas de processamento paralelo [Ghodsi et al., 2011].



(a) Twidd (b) Eclat

Figura 5.4: Efeitos sobre desempenho em 5 executores (*significância* = 0.05) para o Twidd e Eclat, respectivamente

Os resultados para o algoritmo Eclat são mostrados na figura 5.4b. Taxas de erro são baixas em todos os modelos, até nos referentes à métrica de tempo de execução, por isso os fatores explicam bem o comportamento em todos os casos. De fato, o *overhead* atribuído à coleta de lixo no Eclat é muito menor do que no Twidd, devido a

sua implementação ser baseada em estruturas simples e tipos primitivos. O resultado do *overhead* reduzido é a diminuição da variabilidade dos tempos de execução em uma mesma configuração, estabilizando o erro geral do modelo.

As quantidades de memória e *cores* em separado continuam a ter importância para o modelo de %GC. Entretanto, o número de partições é o fator que mais influencia o tempo de execução: 79% da variação. Esse fato é consistente com o compromisso discutido na seção 4.3 entre o tamanho das t-lists e a quantidade de bases locais utilizadas no Eclat, sendo que a quantidade é determinada pelo número de partições (*# partições*).

Analisando esses resultados em termos das dimensões de diagnóstico *Formato de dados* e *Paralelismo adequado* definidas na seção 3.3, podemos destacar as seguintes considerações:

Formato de dados Excessiva atividade com coleta de lixo gera incertezas ao avaliar o tempo de execução de aplicações, principalmente as que manipulam tipos de dados complexos, como é o caso do Twidd. Em contrapartida, a métrica %GC permanece consistente e produz modelos de boa qualidade (isto é, bem explicados pelos fatores e com erro baixo) mesmo quando o tempo de execução apresenta elevada variação. O impacto da coleta de lixo no Eclat é menor, e o tempo de execução é bem explicado pelos fatores e portanto, pode ser considerado um indicativo preciso do desempenho da aplicação.

Paralelismo adequado Consideráveis 79% das variações do tempo de execução sendo explicadas pelo número de partições juntamente com pouco *overhead* de coleta de lixo sugerem que o Eclat é um bom candidato para uma investigação detalhada no quesito nível de paralelismo.

5.3.2 Escalonamento de tarefas

Estudamos em seguida os comportamentos de Twidd e Eclat em vários níveis de paralelismo (número de partições). Os níveis foram escolhidos levando-se em conta a quantidade de *cores* disponíveis no *cluster* (72) e em execuções adicionais necessárias para explicar casos pontuais, discutidos em seguida. A figura 5.5 mostra os resultados. No Twidd variamos o paralelismo do primeiro passo de balanceamento do algoritmo, responsável por fundir pares prefixo/μ-árvores (figura 4.1, *# partições(1)*). No Eclat variamos a quantidade de partições da entrada, que tem relação com o tamanho das t-lists resultantes do passo inicial (figura 4.3, *# partições*). Os demais pontos de adap-

tação e configurações de ambiente em ambas aplicações foram mantidos fixos, para efeito de comparação.

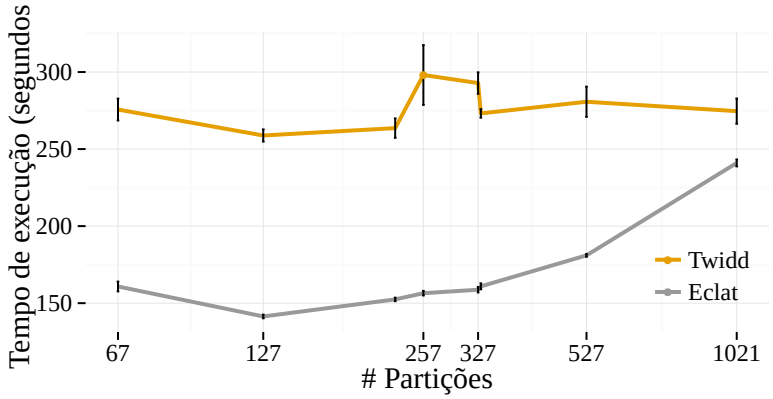


Figura 5.5: Em alguns casos, encontrar o nível adequado de paralelismo não é uma questão de aumentar/diminuir o número de partições, apenas. O degrau ente 257 e 327 partições no Twidd mostra que fatores além do número de partições afetam o seu desempenho.

O particionamento no Eclat tem o efeito que se espera, ou seja, existe um ponto ideal que representa o melhor do compromisso entre o número de tarefas e seus respectivos custos. Nesse caso, encontramos que 127 partições produzem os melhores resultados das configurações consideradas; menos ou mais partições degradam o desempenho da aplicação. Além disso, no caso do Eclat, mais partições implicam em mais t-lists replicadas, fato que tende a aumentar a utilização de memória.

Ao avaliar os resultados obtido com o Twidd, concluímos que estimar um bom particionamento pode não ser tão possível simplesmente variando o número de partições de forma arbitrária. O tempo de execução para o Twidd se comporta de forma similar ao observado com o Eclat inicialmente. Entretanto, no intervalo entre $[227, 331]$ notamos um fenômeno interessante: o tempo de execução aumenta com 257 e 331 partições. Além disso, observamos uma alta variação nos resultados com 257 partições (*questão 1*) e um abrupto ganho em desempenho ao aumentar o número de partições de 327 para 331 (*questão 2*). Esse comportamento peculiar nos levou a analisar de forma mais cuidadosa cada um dos pontos anômalos, de forma a esclarecer a causa dessas variações.

Uma análise detalhada dos logs de execução para 257 partições (*questão 1*) nos permitiu diagnosticar a fonte da alta variação naqueles experimento. Primeiramente, isolamos o estágio causando essa anomalia (o quinto estágio, que executa o algoritmo FPGrowth nas árvores finais reparticionadas) para observar suas tarefas ao longo do tempo em função de seus tempos de execução. Dividimos ainda mais a análise em

melhores e piores cenários de uma mesma variação, ou seja, as replicações de um experimento realizado sob as mesmas condições. A figura 5.6 ilustra o melhor cenário encontrado. Os tempos de execução permanecem entre 10 e 30 segundos na maioria das tarefas, exceto para duas exceções que demoram aproximadamente 60 e 40 segundos. Além disso, os tempos de coleta de lixo (GC) das tarefas se mantém coerente com os tempos de execução, isto é, tarefas que passam mais tempo coletando lixo foram aquelas que executaram por mais tempo.

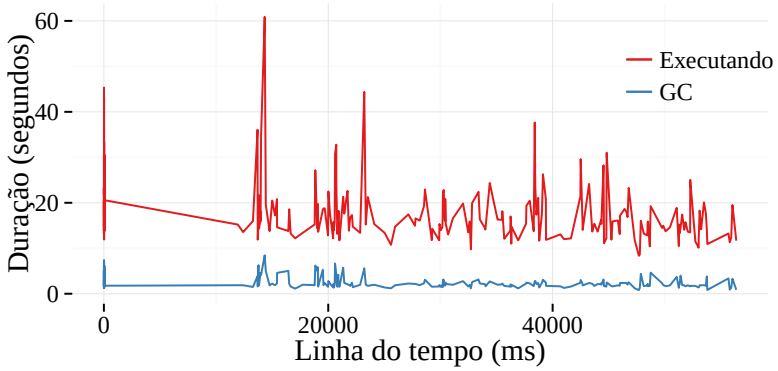


Figura 5.6: Melhor cenário do quinto estágio do Twidd. Mesmo com desbalancemento nos tempos de execução, não observamos o pior caso da figura 5.7

O pior cenário aparece na figura 5.7. Aqui destacamos as tarefas que causaram maior aumento no tempo de execução (72 dentre 257 tarefas). Ao observar o eixo y nas figuras 5.6 e 5.7 notamos que nem mesmo a tarefa mais custosa do melhor cenário chegou perto do limite superior resultante de 8 tarefas no pior cenário (≈200 segundos). De fato, tempos agregados atribuídos à coleta de lixo de mais de 1 minuto estão relacionados a essas 8 tarefas. Completamos nosso diagnóstico ao observar que as mesmas 8 tarefas foram escalonadas quase ao mesmo tempo (em um intervalo de 30ms) no mesmo executor do *cluster*. Isso é consistente com nossa análise da seção 5.3.1, na qual indicamos que altas variações nos tempos de execução podem ser explicadas por *overhead* em coleta de lixo. Além disso, esse comportamento expõe a sensibilidade do particionamento a tarefas retardatárias (*questão 2*), pois quatro tarefas foram capazes de melhorar o desempenho em $6,7\%$ (≈19 segundos) em média ao eliminar alguns dos *outliers* e reduzir flutuações nos tempos de execução (para 331 partições, figura 5.5).

Analisando esses resultados em termos da dimensão de diagnóstico *Localidade de tarefas* definida na seção 3.3, podemos destacar as seguintes considerações:

Localidade de tarefas Aplicações irregulares são especialmente sensíveis ao nível de paralelismo e localidade de tarefas. No Twidd, identificamos casos onde a co-alocação

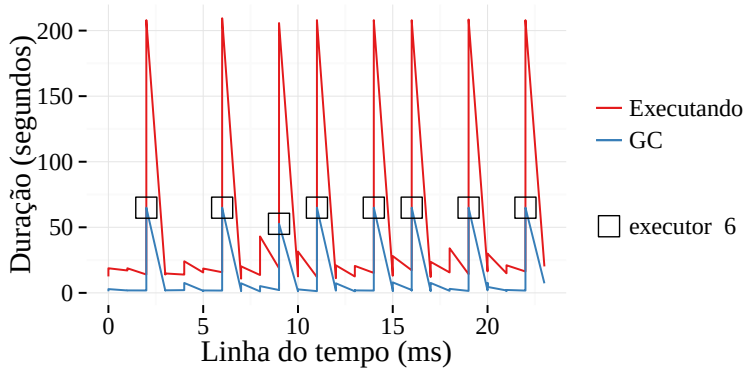


Figura 5.7: Pior cenário para o quinto estágio do Twidd. Tarefas custosas são co-alocadas no executor 6, causando *overhead* na coleta de lixo e afetandoa execução global

de tarefas custosas em executores já sobrecarregados pode causar degradações de desempenho consideráveis. O cenário mais típico de executores se comportando inesperadamente lentos acontece quando os mesmos se encontram em intervalos de intensa coleta de lixo. Por esse motivo, escalonadores para sistemas de processamento massivo poderiam se beneficiar de informações sobre o estado da coleta de lixo nas máquinas do *cluster*. Além disso, flutuações e discrepâncias muito grandes nos tempos dedicados à coleta de lixo por tarefa são um indício de que alguns executores podem estar sendo sobrecarregados.

5.3.3 Execução adaptativa

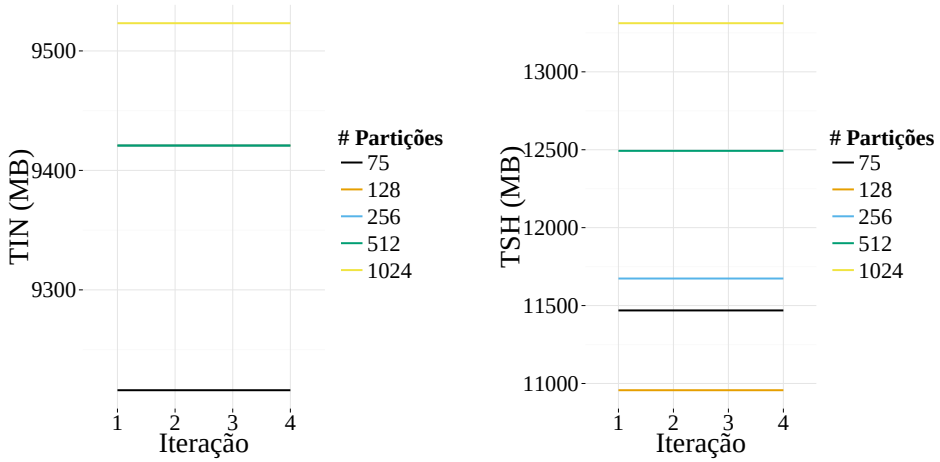
Muitos algoritmos de aprendizado de máquina ou processamento de grafos com condições de terminação que dependem de uma convergência. Quando traduzidas para a representação de DAGs do Spark, essas técnicas expõem oportunidades de otimização de desempenho baseada em re-execuções. De fato, a estratégia de particionamento pode ser alterada no início de cada estágio, pois os dados são reorganizados a cada *shuffle*. No caso de algoritmos iterativos, a lógica de uma iteração certamente será aplicada novamente em iterações posteriores. Assim, ao associar a lógica de programas paralelos com os dados que os mesmos produzem/consomem, podemos classificar grupos de estágios em:

1. *Equivalentes* (P_{eq}), que executam a mesma função sobre os mesmos volumes de entradas/saídas.

2. *Similares* (P_{sim}), que executam a mesma função sobre volumes diferentes de entradas/saídas.

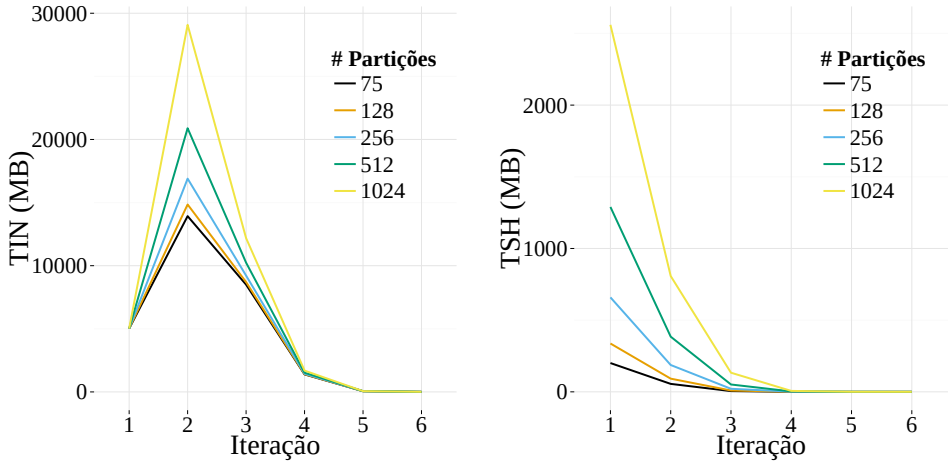
3. *Não relacionados* (P_{un}), que compartilham poucas propriedades ou nenhuma.

A abordagem adotada pelo PageRank produz o mesmo padrão de comunicação a cada iteração: vértices do grafo compartilham rankings entre vizinhos. Por isso, os estágios entre iterações são do tipo P_{eq}. A figura 5.8 mostra o progresso constante dos parâmetros de entrada do PageRank em função das iterações em diferentes níveis de paralelismo.



(a) Tamanho da entrada (b) Leitura de shuffle

Figura 5.8: PageRank: O tamanho do problema permanece constante



(a) Tamanho da entrada (b) Leitura de shuffle

Figura 5.9: Eclat: O tamanho do problema é variável

Identificamos também algoritmos que re-executam as funções de um estágio a cada iteração, mas em contextos diferentes. Nesse caso, o problema tem o potencial

para crescer/encolher a cada iteração. Mesmo assim, um histórico de execuções podem auxiliar o escalonador na reconfiguração de iterações futuras. O Eclat, por exemplo, gera diferentes contextos a cada iteração, devido à sua natureza combinatória. Por isso, suas iterações são P_{sim}. A figura 5.9 mostra a evolução dos parâmetros de entrada do Eclat em função das iterações da execução, para diferentes níveis de paralelismo (particionamento na leitura das transações). O tamanho da entrada (t-lists) tem o mesmo comportamento ao longo das iterações, crescendo para gerações iniciais de itemsets e diminuindo para gerações posteriores. Leituras de *shuffle* sempre diminuem pois se referem à agregação global de contagens locais, sendo inversamente proporcional ao número de itemsets infrequentes descobertos a cada ciclo do algoritmo.

Otimizando P_{eq} Se encontrarmos a quantidade ideal de paralelismo em iterações iniciais, podemos utilizar esse conhecimento para acelerar a execução de iterações futuras. Aplicamos essa otimização manualmente para o PageRank. A figura 5.10 mostra os resultados. Inicialmente, executamos o algoritmo em diferentes configurações de número de partições. Nos referimos a esses resultados como *execução conservativa* porque o número de partições da entrada (derivado do número de blocos do arquivo no HDFS) é utilizado como o paralelismo padrão para as reduções em todas as iterações. Então, ao identificar que 1024 partições aceleraria as reduções, re-executamos os testes utilizando diferentes quantidades para o número de partições de entrada, porém limitando o paralelismo da redução em 1024. Nos referimos a esses novos resultados como *execução adaptativa* pois independentemente do número de partições de entrada (nas quais o usuário tem pouco ou nenhum controle já que depende do ambiente de armazenamento), somos capazes de otimizar novos passos de acordo com o seu custo inerente observado historicamente. Assim, fomos capazes de aprimorar o desempenho geral, atingindo ganhos (*speedups*) de $2,2$x, $1,7$x e $2,3$x nas primeiras três configurações. Os resultados com 1024 partições são exatamente os mesmos, pois a execução foi utilizada como *baseline* para o ajuste das outras execuções. Finalmente, nenhum ganho é observado quando o número de partições de entrada excede o custo das reduções (com 2048 partições), pois o último passa a não representar mais um gargalo no desempenho.

Otimizando P_{sim} Quando o tamanho do problema varia entre iterações, seria errôneo assumir que o mesmo paralelismo adotado inicialmente garantiria os mesmos ganhos ao longo do tempo. Aplicar, portanto, a mesma estratégia adaptativa adotada para o PageRank no Eclat seria uma falta de critério. A aplicação deveria ser capaz de estimar novas configurações baseado-se em algum conhecimento comum de execuções anteriores. Nesse contexto, o conhecimento comum é que a aplicação executou o mesmo

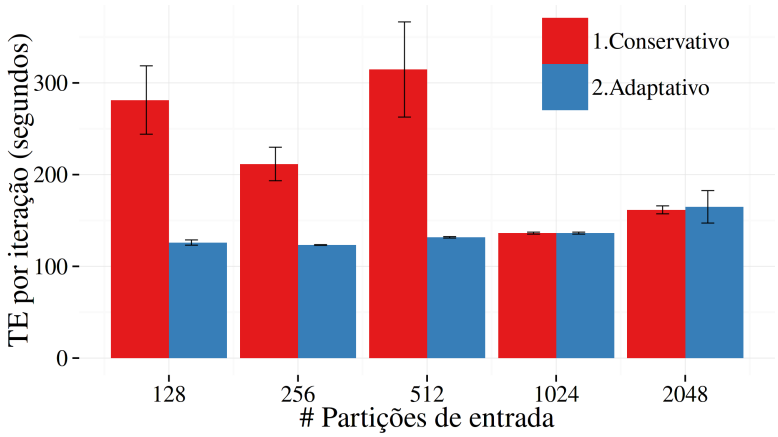


Figura 5.10: Otimizando P_{eq}. Estágios se repetem com o mesmo padrão de comunicação

estágio sob certas circunstâncias (entradas e saídas, como ilustrado na figura 3.5) e teve um custo associado (tempo de execução nesse caso). Então, o processo de otimização se torna uma questão de, dado um novo conjunto de entradas e saídas e o conhecimento histórico, qual seria o ajuste de desempenho apropriado?

Nós validamos essa ideia ao criar uma heurística simples aplicada no *ponto de adaptação* na figura 4.3. Observamos o fator de seletividade do primeiro operador antes que o *shuffle* ocorra, isto é, o valor $selec = \frac{input}{shuffleWrite}$. Esse fator é utilizado para ajustar o paralelismo para a redução da primeira iteração. Configuramos então o número de partições para $nparts = \max(\lceil \frac{inputPartitions}{selec} \rceil, totalCores)$. O uso da função de máximo sobre $totalCores$ tem o objetivo de prevenir subutilização de *cores* logo no início do processo, evitando que o efeito elástico não seja levado em conta. O restante das iterações são ajustadas proporcionalmente a $shuffleWrite$ e $nparts$ das iterações anteriores. O algoritmo 1 apresenta os detalhes desse procedimento. Nesse procedimento, consideramos que a primeira chamada será com $k = 0$, representando o número de partições de entrada utilizada de forma conservativa na execução original. Assim, chamadas subsequentes do procedimento definem o número de partições a ser configurado para as iterações da nova execução, isto é, para a primeira iteração ($k = 1$), para a segunda iteração ($k = 2$) e assim sucessivamente.

Por exemplo, considere um cenário com 5017,6 MB de dados de entrada, 1289,6 MB de escrita de shuffle, 512 partições no primeiro estágio da primeira iteração e um ambiente de execução com 72 cores no total. Dessa forma, a primeira chamada do ajuste seria:

`get-num-partitions(`$nparts_0$ `= 512, 5017.6, 1289.6, 72) = 132`

Em particular, o fator de seletividade para esse contexto seria $selec = \frac{5017,6}{1289,6} =$

Algoritmo 1 Heurística para o ajuste manual do Eclat

1: **Função:** GET-NUM-PARTITIONS($nparts_k, input, shuffleWrite, totalCores$)
2: $selec \leftarrow \frac{input}{shuffleWrite}$
3: $nparts_{k+1} \leftarrow \lceil \frac{nparts_k}{selec} \rceil$
4: **if** $(k+1) = 1$ **then**
5: $nparts_{k+1} \leftarrow \text{MAX}(nparts_{k+1}, totalCores)$
6: **return** $nparts_{k+1}$

$3,89$ e o número de partições escolhidas para a primeira iteração seria $nparts_1 = \lceil \frac{512}{3,89} \rceil = 132$. Da mesma forma, a segunda chamada para a segunda iteração que escreveu $384,9$ MB no *shuffle* e recebeu como entrada a quantidade de dados da escrita de *shuffle* da iteração anterior ($1289,6$ MB) seria equivalente à chamada:

`get-num-partitions(`$nparts_1$ `= 132, 1289.6, 384,9, 72) = 40`

Nesse caso, o cálculo realizado seria $nparts_2 = \lceil \frac{132}{\frac{1289,6}{384,9}} \rceil = \lceil \frac{132}{3,35} \rceil = 40$. De fato, o mesmo raciocínio pode ser usado para quantas iterações forem necessárias durante a execução. Portanto, o grau de paralelismo acompanharia a elasticidade natural do algoritmo ao longo das iterações.

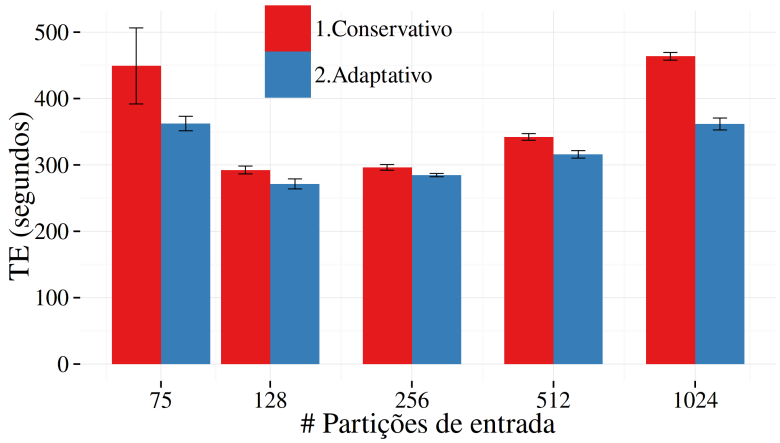


Figura 5.11: Otimizando P_{sim}. Estágios se repetem com diferentes entradas.

A figura 5.11 mostra os resultados da aplicação dessa heurística. Novamente, *execução conservativa* se refere a manter o mesmo número de partições em todas as iterações e *execução adaptativa* é o resultado da aplicação da heurística descrita. A abordagem adaptativa é especialmente eficiente em consertar estimativas extremistas do número de partições (75 e 1024 partições de entrada). O ganho é pequeno em configurações intermediárias devido à aproximação do paralelismo ótimo, no qual a geração de candidatos (mapeamento) domina o custo da contagem global (redução). A execução adaptativa tem também o efeito de reduzir a quantidade de recursos utilizados

a cada passo, reforçando a busca por um *paralelismo adequado*. A figura 5.12 indica a redução na utilização de recursos em relação à abordagem conservativa. O número de tarefas disparadas é reduzido em pelo menos 78,4% em todas as configurações, mesmo nos casos onde os ganhos em tempo de execução são modestos. Vale lembrar que na figura 5.5 que o comportamento esperado para o Eclat é que haja um nível ideal de paralelismo, que refletiria o compromisso entre a replicação de t-lists e seus tamanhos.

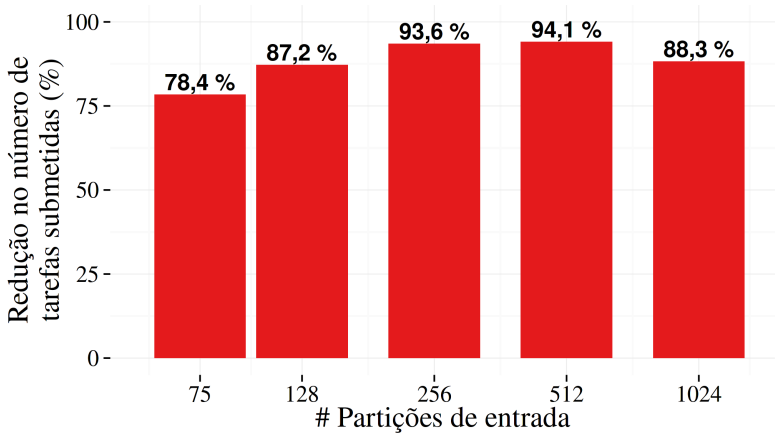


Figura 5.12: Otimizando P_{sim}. Porcentagens indicam redução na utilização de recursos da abordagem adaptativa em relação à conservativa *(1.Convervativo)*, indicada na figura 5.11.

Analisando esses resultados em termos da dimensão de diagnóstico *Paralelismo adequado* definida na seção 3.3, podemos destacar as seguintes considerações:

Paralelismo adequado Em aplicações iterativas (mas não apenas nelas), há sempre re-execução de sub-DAGs representando passos da computação, sujeitos ou não ao mesmo contexto. Otimizadores de desempenho devem ser capazes de adaptar uma execução de acordo com padrões de comunicação observados anteriormente. Além de ganhos em desempenho, as estratégias empregadas contribuem para um paralelismo adequado, ou seja, aquele que usa a quantidade ideal de recursos, nem mais, nem menos.

5.3.4 Balanceamento de carga

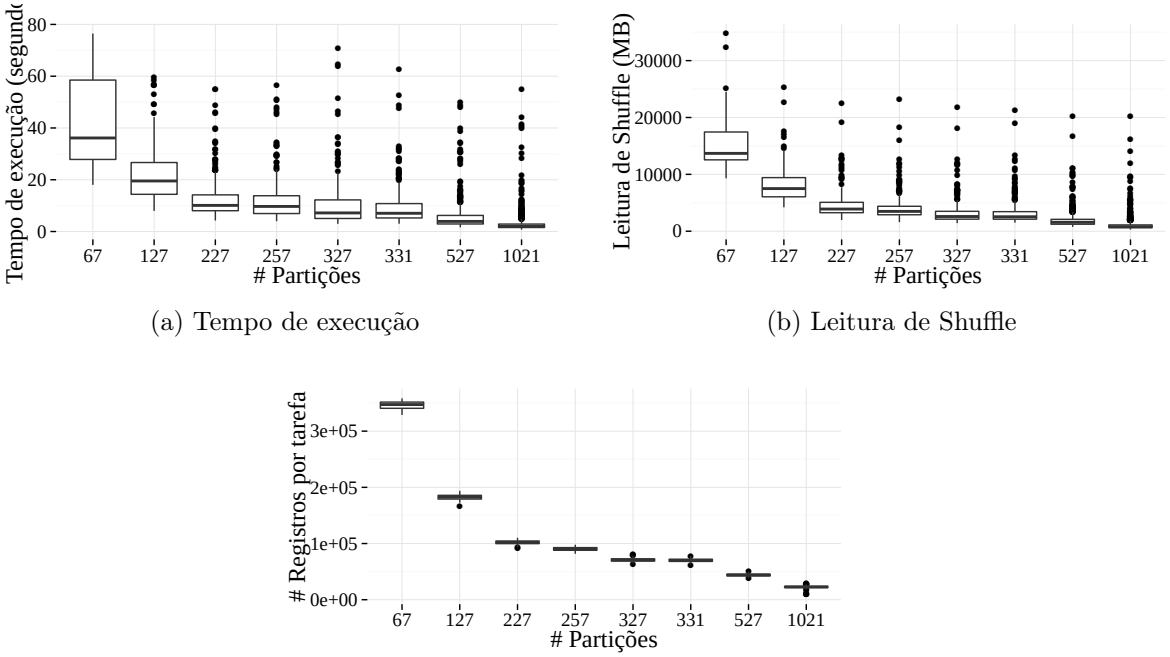
Abordamos a dimensão de balanceamento de carga ao investigar detalhadamente o quarto e quinto estágios do Twidd (figura 4.1: *# partições (1)* e *# partições (2)*, respectivamente). Nesse ponto, revisitamos os testes realizados na seção 5.3.2 com

um foco em efeitos relacionados ao balanceamento de carga ao invés de desempenho isolado.

A figura 5.13 apresenta os resultados para métricas adicionais referentes ao quarto estágio. O eixo x representa o número de partições do estágio. O desbalanceamento nos tempos de execução tende a aumentar à medida que mais partições são utilizadas. Além disso, ao observarmos a quantidade de leitura de *shuffle* encontramos um comportamento parecido, isto é, a quantidade de dados que cada tarefa lê do *shuffle* também se torna mais desbalanceada com o aumento do nível de paralelismo. Esses resultados sugerem que o tempo de execução cresce juntamente com o volume de dados da etapa de leitura do *shuffle*. De fato, as correlações de Pearson e Spearman entre essas duas dimensões são $0,94$ e $0,95$ respectivamente. Entretanto, encontramos quase nenhum *outlier* no número de registros processados por tarefa, significando que o particionador está dividindo a carga de trabalho igualmente no que se refere ao número de registros escritos no *shuffle* pelo estágio anterior. Por isso, o desbalanceamento se dá pelo fato de que os registros têm diferentes tamanhos e, portanto, custos de processamento variáveis.

Em suma, aumentar o número de partições de um estágio nem sempre consegue eliminar o desbalanceamento da carga de trabalho. Em casos onde isso não é suficiente, o particionamento baseado em *hash* (estratégia padrão no Spark) não consegue capturar e distribuir a carga igualmente e, portanto, um aumento arbitrário no número de partições pode resultar em casos anômalos (seção 5.3.2). Isso acontece porque a estratégia de particionamento baseada puramente em *hash* é ideal quando o número de registros é uma boa estimativa para o custo da tarefa, isto é, se todo registro tivesse aproximadamente o mesmo custo para processamento. Entretanto, isso pode não ser verdade em aplicações como o Twidd, em que os registros possuem tamanho variável e representam estruturas complexas. Mesmo com particionamento por intervalos (*range partitioning*), cuja estratégia é dividir registros igualmente de acordo com as frequências das chaves, não seria suficiente para capturar o desbalanceamento que existe entre registros de uma mesma chave. Nesses casos extremos, a única alternativa viável seria a incorporação de instrumentações avançadas para identificar a distribuição dos dados levando em conta os custos e possibilitando uma divisão mas inteligente da carga de trabalho.

Uma conclusão diferente é encontrada pela análise do quinto estágio do Twidd. Os resultados são mostrados na figura 5.14. A mediana não é uma boa estimativa para tempo de execução em nenhum dos níveis de paralelismo, como podemos observar pela figura 5.14a. Nesse ponto, não podemos afirmar se o custo de processamento variável acontece devido a um particionamento ruim ou se isso reflete um comportamento ine-



(a) Tempo de execução

(b) Leitura de Shuffle

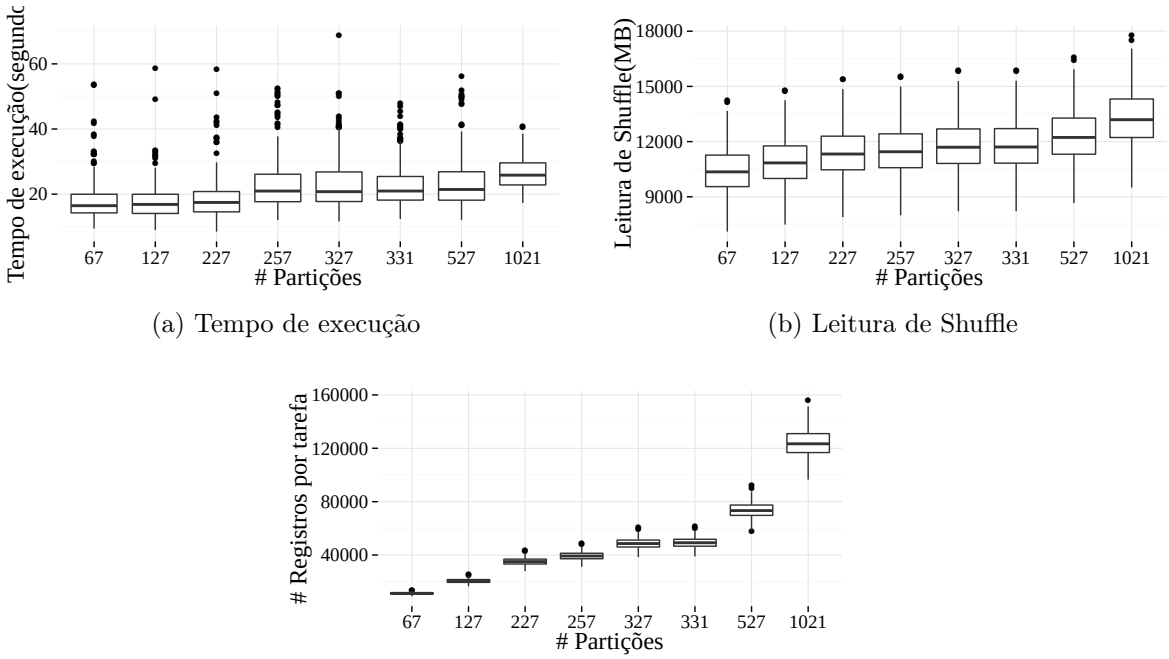
(c) Registros

Figura 5.13: Quarto estágio de execução do Twidd. A maioria do desbalanceamento pode ser explicado pela leitura desigual de *shuffle*. Correlações entre leitura do *shuffle* e tempos de execução: Pearson = 0,94; Spearman = 0,95

rente do algoritmo em questão. Entretanto, olhando para as distribuições do volume de leitura do *shuffle* (figura 5.14b) e do número de registro processados por cada tarefa (figura 5.14c) concluímos que, por outro lado, a mediana representa essas duas métricas (as caixas são simétricas, exceto por alguns *outliers* na quantidade de registros processados). Então, a causa do desbalanceamento nos tempos de execução do quinto estágio não pode ser devido a um particionamento ruim. Pelo contrário, é uma característica inerente do algoritmo. Essa asserção faz mais sentido ao levarmos em conta a computação realizada nesse estágio: FPGrowth, cuja complexidade é proporcional ao número de itens únicos em cada FPTree final e ao limiar de suporte que atua como um critério de poda. Esses dois fatores em conjunto aumentam o potencial para custos desbalanceados entre tarefas, mas sem relação nenhuma com o tamanho da entrada.

Analisando esses resultados em termos das dimensões de diagnóstico definidas na seção 3.3, podemos destacar as seguintes considerações:

Balanceamento de carga O desbalanceamento nos custos de processamento pode ser inerente ao procedimento sendo executado ou resultante de um particionamento ruim. O número de registros não geram sempre boas estimativas do custo computacional das



(a) Tempo de execução

(b) Leitura de Shuffle

(c) Registros

Figura 5.14: Quinto estágio de execução do Twidd. A maioria do desbalanceamento é inerente do algoritmo. Correlações entre leitura do *shuffle* e tempos de execução: Pearson = 0,19; Spearman = 0,23

tarefas e isso acontece quando o custo de processamento de registros em uma aplicação é variável. Uma alternativa para esses casos é contar com mecanismos avançados de instrumentação, capazes de estimar a distribuição das chaves juntamente com seus custos agregados e auxiliar o particionador a tomar decisões mais inteligentes. Por outro lado, detectar tais comportamentos e relatá-los ao usuário do sistema pode ser útil no caso da necessidade de uma intervenção manual.

Capítulo 6

Ferramenta de Reconfiguração

Como visto anteriormente, o ajuste do particionamento ao longo da execução de uma aplicação é um fator determinante no desempenho de programas paralelos. Entretanto, do ponto de vista prático, ajustar a implementação de uma aplicação pode ser uma tarefa custosa que pode exigr um engajamento exagerado do usuário. Além disso, o perfil de usuários típicos desses ambientes não é de operadores de sistemas que se preocupam com o desempenho, mas sim de analistas e/ou cientistas de dados que se interessam mais pelas técnicas dos algoritmos e domínio do problema. Assim, nosso último objetivo neste trabalho é desenvolver uma ferramenta que facilite esse processo de otimização.

Três requisitos guiaram o projeto da ferramenta de reconfiguração: *(1)* a ferramenta deve ser o mais agnóstica possível em relação ao versionamento do ambiente e implementação; *(2)* a ferramenta deve expor um modelo conceitual simples o bastante para o fácil entendimento dos efeitos dos ajustes na aplicação; e *(3)* a ferramenta deve ser extensível, permitindo que estratégias especializadas possam ser incorporadas com o mínimo de esforço. Aplicamos essa ideia no contexto de aplicações recorrentes, ou seja, aquelas que são re-executadas frequentemente em um *workflow* de análise de dados.

A figura 6.1 representa a arquitetura da ferramenta desenvolvida neste trabalho. A aplicação do usuário inicia com ou sem informação de logs de execuções passadas *(0)*. Por comodidade, utilizamos o formato de logs padrão disponível do Spark, que contém eventos e métricas associadas com a execução (seção 5.2). Caso a informação de logs não seja fornecida, a aplicação executa normalmente e registra o primeiro log, que poderá ser utilizado como entrada para futuras execuções. Em seguida, o sistema instancia internamente um *Adaptation Helper* (seção 6.2) *(1)*, que por sua vez inicializa o *Analyzer* (seção 6.1) *(2)* para analisar o log e tomar decisões de reajuste baseadas nas estatísticas extraídas e políticas configuradas. Essas decisões são retornadas ao

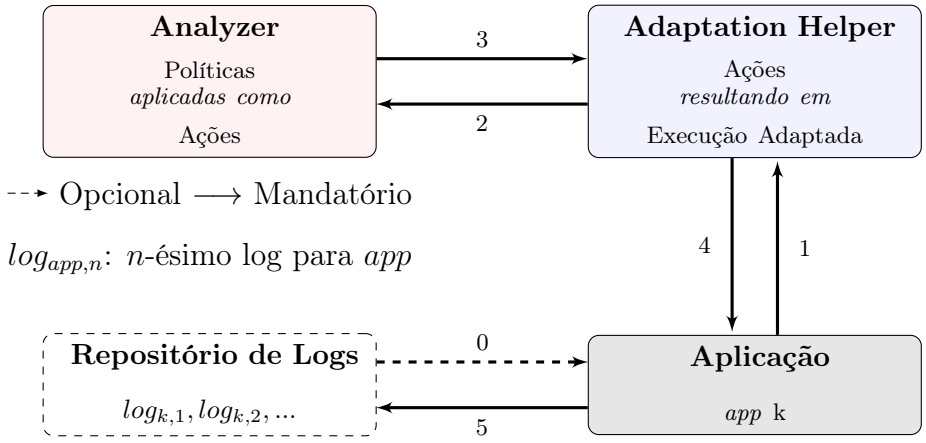


Figura 6.1: Arquitetura da ferramenta de reconfiguração

Adaptation Helper na forma de *ações* a serem aplicadas na execução atual *(3)*. O *Adaptation Helper* então traduz as ações em configurações e/ou RDDs propriamente particionados, que são retornados para a aplicação *(4)*. Por fim, o log da nova execução é registrado e armazenado no repositório *(5)*. Isso finaliza um ciclo de reconfiguração, que pode ser re-executado enquanto a aplicação apresentar gargalos de desempenho.

6.1 Analyzer

O propósito do *Analyzer* é receber logs de execução do *Adaptation Helper*, identificar oportunidades para o ajuste do particionamento, aplicar as políticas configuradas nos respectivos pontos e finalmente entregar um conjunto de ações para o *Adaptation Helper*. Em particular, as oportunidades de ajuste são identificadas através dos *pontos de adaptação* (seção 3.2.2), que indicam fronteiras nas quais a presença de comunicação global permite a alteração do particionamento. Ações nesse contexto indicam *ajustes no particionamento* em pontos da aplicação ou *configurações globais*, que devem ser aplicadas estaticamente antes do início da execução propriamente dita. O *Analyzer* também captura o padrão de comunicação da aplicação, isto é, suas decisões levam em conta se a aplicação é regular/irregular e/ou iterativa/não-iterativa. Para tanto, empregamos as mesmas premissas e comportamentos discutidos no capítulo 5. A versão atual suporta cinco tipos de ações:

- `UConfAction`: altera uma *configuração global*;
- `UNPAction`: atualiza o número de partições em um *ponto de adaptação*;
- `UPAction`: atualiza o particionador em um *ponto de adaptação*;

- `WarnAction`: inclui um aviso nos logs de usuário como um esforço em lidar com os casos onde alguma fonte de ineficiência foi detectada mas nenhuma ação automática para mitigar o problema é conhecida;
- `NoAction`: indica que nenhuma ação deve ser tomada.

Por exemplo, `UPAction` pode solicitar a alteração do particionador de *hashPartitioner* para *rangePartitioner*. A ação *WarnAction* pode destacar para o usuário algum problema que requer sua intervenção manual, como a identificação de fontes de desbalanceamento (5.3.4). Por outro lado, uma ação do tipo `UConfAction` consegue, por exemplo, alterar a estratégia de serialização dos RDDs ao modificar uma configuração global (`spark.serializer`) para a aplicação como um todo.

O plano de ações depende do conjunto de políticas configuradas no *Analyzer*. Modelamos as políticas com assinaturas bem definidas, recebendo informação do ambiente e/ou ponto da execução e retornando necessariamente uma ação, ou a ausência da mesma através do `NoAction`. Mais especificamente, dividimos as políticas em duas categorias: *políticas da aplicação*, que consideram a execução como um todo e modificam as configurações globais através da ação `UConfAction`; e *políticas de particionamento*, que atuam sobre um ponto de adaptação através das ações `UNPAction`/`UPAction` e recebem tanto informações globais quanto específicas do ponto em questão. Detalhes de quais informações estão disponíveis para a implementação em cada um dos tipos de políticas estão presentes na seção 6.3.2, onde discutimos o modelo da interface exportada para o usuário. Para efeitos de amostra, um conjunto de políticas de particionamento utilizado na avaliação experimental é descrito na seção 6.3.4; além disso, discutimos detalhadamente a motivação e a criação de uma política de aplicação no contexto de localidade de dados na seção 6.3.3.

6.2 Adaptation Helper

O *Adaptation Helper* une aplicações de usuário com o módulo de tomada de decisão, o *Analyzer*. A sua função principal é traduzir ações retornadas pelo *Analyzer* em modificações reais na execução corrente, de forma transparente para o usuário. Em resumo, através da API fornecida (seção 6.3), o usuário pode alimentar o ambiente com o caminho de uma execução anterior, preadaptar e/ou incluir novas políticas e chamar operadores de RDD cientes dos pontos de adaptação.

O usuário interage com o *Adaptation Helper* através de operadores Spark sobrescritos, que entendem a semântica dos pontos de adaptação. Os operadores que marcam pontos de adaptação são bem definidos no sistema (*reduceByKey*, *textFile*, *join*, etc.),

ou seja, qualquer operador que permita a configuração do número de partições do RDD e/ou seu partitionador. Nosso sistema utiliza um mecanismo da linguagem Scala chamado *implicits* que é capaz de redefinir os operadores padrão do ambiente. Por esse motivo, o único requisito para a utilização do framework é explicitamente importar suas definições (*AdaptableFunctions*).

O *Adaptation Helper* pode ser instanciado (atomicamente) em duas situações. Primeiro se um operador sobrescrito for chamado pela primeira vez; segundo, se o programa do usuário chamar `preAdapt` inicialmente para incluir políticas customizadas, por exemplo. Ao iniciar, o *Adaptation Helper* dispara a inicialização e execução do *Analyzer*. Nesse momento, o *Adaptation Helper* potencialmente tem todas as ações retornadas pelo *Analyzer* associadas com os respectivos pontos de adaptação.

Dado que as ações estão disponíveis, toda vez que o código do usuário executar algum operador sobrescrito o sistema irá primeiramente verificar se existe alguma açãoque precisa ser aplicada no ponto corrente. Se esse for o caso, um novo RDD é criado a partir da semântica da ação, como atualização do número de partições ou particionador. Caso contrário, a execução continua com o comportamento padrão esperado. Essa característica faz com que a solução seja totalmente segura no sentido de não inutilizar implementações antigas.

Por fim, foi preciso estabelecer duas decisões de projeto: *(1)* como os logs são encaminhados para o *Analyzer* e *(2)* como os pontos de adaptação são diferenciados entre si. Para a primeira decisão, escolhemos utilizar um recurso já presente no Spark, que é a possibilidade de adicionar configurações com nomes genéricos através do comando *SparkConf*. Assim, a configuração `spark.adaptive.logpath` deve ser apontada para o caminho onde o log de uma execução anterior pode ser encontrado. Para a segunda decisão, existem duas alternativas: *(a)* para os operadores sobrescritos, adicionamos um parâmetro opcional que é o identificador do *ponto de adaptação*, na forma de um nome qualquer; *(b)* se nenhum nome for especificado pelo usuário, o sistema constrói um nome padrão composto do nome do operador e a linha do código onde o mesmo foi chamado. Como trabalhos futuros, planejamos incluir um mecanismo automático de associação entre aplicações, logs de execuções passadas e respectivos pontos de adaptação.

6.3 API da ferramenta

A interface de utilização da ferramenta incorpora as funcionalidades do *Analyzer* e do *Adaptation Helper*. A figura 6.2 descreve a API da ferramenta. Em resumo, o usuá-

rio pode alimentar o ambiente com o caminho de uma execução anterior (*Feedback de Log*), preadaptar e/ou incluir novas políticas (*Inicialização*), invocar operadores de RDD cientes dos pontos de adaptação (*Operadores Sobrescritos*) e estender políticas específicas para um novo gargalo de desempenho ou de acordo com características específicas de domínio (*Criação de Políticas*). Em seguida descrevemos a estrutura básica de um programa (seção 6.3.1), o modelo utilizado pela API da ferramenta (seção 6.3.2) e o procedimento para criação de novas políticas (seção 6.3.3). Finalmente, apresentamos na seção 6.3.4 as políticas-exemplos que serão exercitadas durante a avaliação experimental.

Feedback de Log: `conf.set("spark.adaptive.logpath",` $log_{app,n}$`)`

Inicialização `preAdapt(`sc`,` p_1`,` p_2`, ...)`

Operadores Re-escritos: `textFile, reduceByKey, aggregateByKey, join, partitionBy`
com um parâmetro adicional *ap* referenciando o ponto de adaptação

Criação de Políticas:
```
MyPolicy extends ApplicationPolicy
MyPolicy extends PartitioningPolicy
```

Figura 6.2: API da ferramenta

6.3.1 Estrutura básica de um programa

Para ilustrar o uso e capacidades da ferramenta, descrevemos um código escrito em Scala para a aplicação clássica *Wordcount*, que possui suporte ao framework em questão. A figura 6.3 representa o código completo, com diferenças em relação a uma implementação comum destacadas em negrito: *(a)* importamos as funções sobrescritas que entendem a semântica dos pontos de adaptação (*linha 1*); *(b)* configuramos o log de uma execução passado através do *SparkConf* (*linha 5*); *(c)* pré-adaptamos a execução antes dos operadores porém sem nenhuma política adicional (*linha 6*); e *(d)* escolhemos explicitamente um nome para o ponto de adaptação da etapa de redução do *Wordcount* (*linha 10*).

É importante mencionar que, de todas as mudanças destacadas, apenas *(a)* e *(b)* são obrigatórias. De fato, o usuário pode escolher não invocar *preAdapt* (*linha 6*) e assim, o *Analyzer* iria analisar o log e gerar o conjunto de ações de forma preguiçosa, isto é, no momento em que o *reduceByKey* fosse chamado. Mais ainda, como o sistema provê um nome padrão para todo ponto identificado como *ponto de adaptação*, o nome *"ap-counting"* na linha 10 também é opcional.

```
1 import br.ufmg.cs.systems.sparktuner.rdd.AdaptableFunctions._
2 object Wordcount {
3   def main(args:  Array[String]) {
4      val conf = new SparkConf().setAppName("Word Count").
5          set("spark.adaptive.logpath", "/tmp/log1")
6      preAdapt(conf)
7      val sc = new SparkContext(conf)
8      val counts = sc.textFile ("/tmp/sample").
9          flatMap (_ split ).map (w => (w,1)).
10         reduceByKey (_ + _, "ap-counting")
11     println (counts.count + "words")
12     sc.stop()
13  }
14 }
```

Figura 6.3: Wordcount adaptável

6.3.2 O Modelo

A interação entre o usuário e a ferramenta pode acontecer em dois níveis. Em um primeiro nível, o usuário está interessado em utilizar as funcionalidades mais básicas e, para isso, basta que ele conheça a API básica para a estruturação de seu programa (seção 6.3.1). Em um segundo nível, o usuário pretende estender uma política específica para sua aplicação e; nesse caso, é importante conhecer como ocorre a interação entre políticas e as informações presentes nos logs de execuções passadas. Com esse propósito, criamos um modelo para representação dos logs de execução (*Modelo de Logs*) e outro para representar tipos de políticas e suas respectivas interfaces de extensão (*Modelo de Políticas*). A figura 6.4 ilustra os detalhes e interações entre esses dois modelos.

Em linhas gerais, o objetivo do *Modelo de Logs* é entregar para o *Modelo de Políticas* informações e estatísticas referentes à aplicação como um todo, representado como o ambiente (*Environment*), ou referentes a um ponto específico, representado como um agregado de estágios e o RDD do *ponto de adaptação* em questão (*AdaptivePointStats*).

O ambiente agrega configurações específicas da máquina virtual sendo utilizada (`systemProperties`), configurações próprias do Spark (`sparkProperties`), recursos disponíveis como *cores* e memória disponível na forma de executores (`executors`) e todos os estágios observados durante a execução. *Stage* representa um agregado de tarefas, que por sua vez contém o conjunto de métricas extraídas dos logs de execução e descritas na seção 5.2. Cada *AdaptivePointStats* identifica e agrega estatísticas (grupo de estágios) referentes a um ponto de adaptação. O RDD identifica um ponto de adaptação através de seu nome, o operador que o gerou e o número da linha de código em que o mesmo foi instanciado. É importante notar que o usuário pode informar

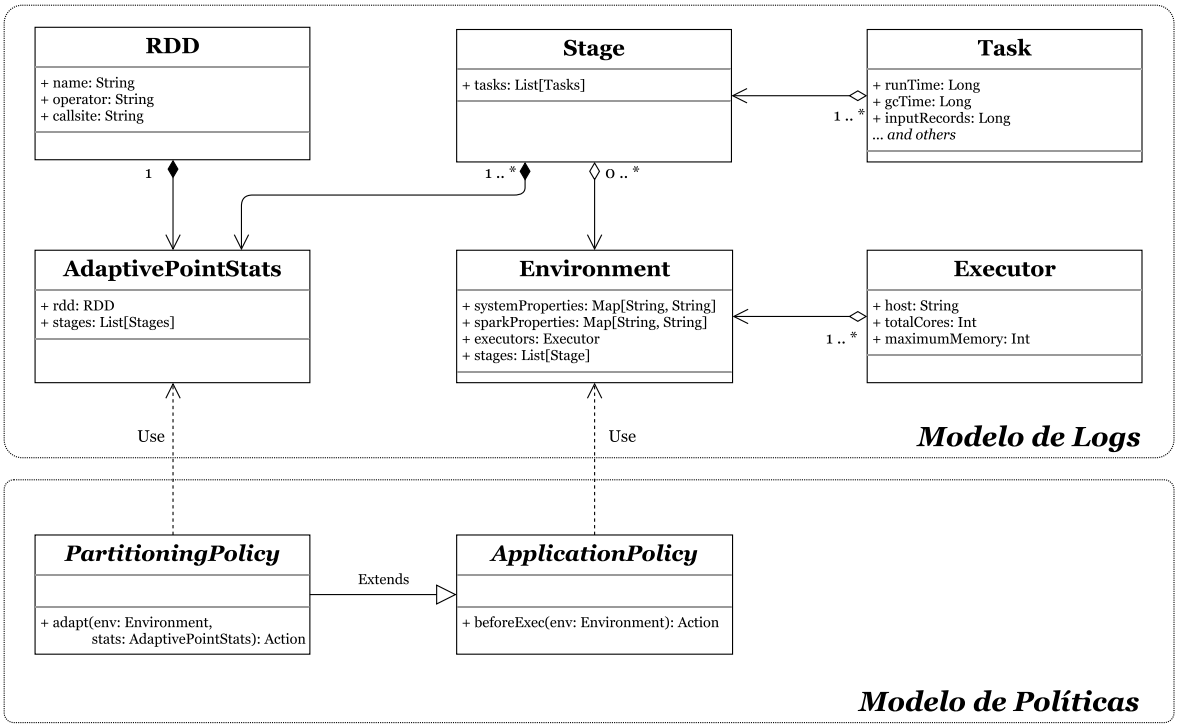


Figura 6.4: Modelo utilizado pela ferramenta de reconfiguração

um nome único para o ponto de adaptação em seu programa, eliminando assim a dependência com relação a linhas de código. Outro ponto pertinente é que nos referimos a um "grupo" de estágios pois o modelo de execução permite que um RDD faça parte de vários estágios: devido à estratégia de *pipelining* entre dependências *estreitas* e à herança de particionamento resultante dessa relação (seção 3.2.1), ações no programa do usuário podem forçar o processamento de qualquer RDD parte desse encadeamento, gerando assim (potencialmente) mais de um estágio por ponto de adaptação.

Identificamos no *Modelo de Políticas* duas categorias: *(1) políticas de aplicação* (*ApplicationPolicy*), que recebe informação global do ambiente e aplica ações de reconfiguração antes que a re-execução se inicie; e *(2) políticas de particionamento* (*PartitioningPolicy*), que pode atuar na aplicação como um todo e além disso, ajustar o particionamento de pontos de adaptação específicos. Na seção seguinte, descrevemos as implicações desse modelo na criação de políticas.

6.3.3 Criando políticas

Para criar uma política de reconfiguração, o usuário deve estender um dos tipos de política descritos no modelo da seção anterior: *ApplicationPolicy* ou *PartitioningPolicy*. Uma vez que esse passo tenha sido realizado, basta incluir a nova política na invoca-

```
1  import br.ufmg.cs.systems.sparktuner._
2  import br.ufmg.cs.systems.sparktuner.model._
3
4  /* Política da Aplicação */
5  object MyApplicationPolicy extends ApplicationPolicy {
6
7    def beforeExec(env:  Environment):  Action = {
8      // retorna uma ação baseada nas estatísticas globais da aplicação
9    }
10 }
11
12 /* Política de Particionamento */
13 object MyPartitioningPolicy extends PartitioningPolicy {
14
15   def beforeExec(env:  Environment):  Action = {
16     // retorna uma ação baseada nas estatísticas globais da aplicação
17   }
18
19   def adapt(env:  Environment, stats:  AdaptivePointStats):  Action = {
20     // retorna uma ação baseada nas estatísticas globais (env) e
21     // específicas do ponto de adaptação (stats)
22   }
23 }
```

Figura 6.5: Interface para criação de *políticas da aplicação* ou *políticas de particionamento*

ção do `preAdapt`, antes da instanciação do *SparkContext*. Várias políticas podem ser incluídas simultaneamente:

`preAdapt(`$conf$`,` p_1`,` p_2`, ...)`,

onde p_i é um objeto ou instância de classe Scala que estende um dos tipos de política.

O modelo para extensão de políticas está ilustrado na figura 6.5. Para criar uma política de aplicação o usuário precisa definir um objeto em Scala que estenda `ApplicationPolicy` e que implemente a função `beforeExec` (*linhas 15-17*). Seguindo a herança apresentada anteriormente, para políticas de particionamento o usuário deve definir um objeto que estenda `PartitioningPolicy` e que implemente as funções `beforeExec` (caso necessário efetuar algum preprocessamento) e `adapt` (*linhas 19-22*).

A figura 6.6 representa o programa *Wordcount* modificado para inclusão de uma política adicional, escrita pelo usuário, chamada `LocalityPolicy`. Mais uma vez, as linhas sublinhadas indicam as diferenças com relação à versão anterior do programa (figura 6.3), isto é, quando apenas as políticas descritas na seção 6.1 estavam sendo utilizadas. A seguir descrevemos a implementação para essa política incluída.

```
1 import br.ufmg.cs.systems.sparktuner.rdd.AdaptableFunctions._
2 object Wordcount {
3   def main(args:  Array[String]) {
4      val conf = new SparkConf().setAppName("Word Count").
5          set("spark.adaptive.logpath", "/tmp/log1")
6      preAdapt(conf, LocalityPolicy)
7      val sc = new SparkContext(conf)
8      val counts = sc.textFile ("/tmp/sample").
9          flatMap (_ split ).map (w => (w,1)).
10         reduceByKey (_ + _, "ap-counting")
11     println (counts.count + "words")
12     sc.stop()
13  }
14 }
```

Figura 6.6: Adicionando uma política na aplicação *Wordcount*

A política `LocalityPolicy` é uma implementação da análise teórica simplificada da estratégia de *Delay Scheduling* [Zaharia et al., 2010], que é aplicada na maioria dos sistemas de processamento massivo de dados, inclusive Spark. Como nosso objetivo é demonstrar a funcionalidade de extensão da ferramenta e não detalhar a análise teórica que apoia a técnica, deixamos a leitura das especificidades para o leitor curioso na publicação original. O objetivo dessa estratégia é aumentar a localidade de dados obtida pelo escalonador de tarefas do ambiente. A ideia é simples e efetiva: como o ganho em executar tarefas com dados locais tem o potencial de ser muito mais eficiente do que buscar dados remotamente para então iniciar o processamento, toda vez que existe uma tarefa para ser escalonada mas nenhum *core* de processamento livre contém o dado da mesma localmente, o escalonador atrasa a atribuição por um intervalo de tempo pré-definido antes de tomar a decisão. A justificativa é que a probabilidade de que uma máquina com dado local libere um *core* cresce à medida que o tempo passa, aumentando assim as chances de escalonar a tarefa localmente. O desafio nesse aspecto, e motivação para uma política de reconfiguração, é determinar o quanto esperar. Esperar pouco significa atrasar a execução das tarefas com pouca garantia de que serão escalonadas localmente. Esperar muito significa que o atraso pode ser maior do que o ganho com a execução local.

Por se tratar de uma *política de aplicação*, a implementação de *LocalityPolicy* exige a extensão de `ApplicationPolicy` e a implementação da função `beforeExec` (figura 6.5, nas linhas 4-10). Mais especificamente, a função recebe as configurações do ambiente relativas à execução passada considerada e deve retornar uma ação que atue sobre a aplicação como um todo. Assim, no contexto da política de localidade,

estamos interessados no tipo de ação `UConfAction`, já que através dela seremos capazes de atualizar a configuração `spark.locality.wait`. O corpo da função implementada deve, portanto, estimar o valor da espera ideal de modo a garantir uma proporção mínima de tarefas locais. Essa proporção mínima funciona como um parâmetro para a política. Por exemplo, encontrar a espera mínima para que, pelo menos, 90% das tarefas sejam escalonadas localmente aos respectivos dados de entrada. Com isso, o mesmo cálculo é feito para cada estágio da aplicação e o valor final de espera é escolhido como sendo o maior dentre esses valores, de modo a garantir que todos os estágios sejam atendidos. O código completo da implementação dessa política é apresentado na figura B.5, contida no apêndice B. Um procedimento similar pode ser usado no desenvolvimento de *políticas de particionamento*, apenas exigindo a implementação adicional da função `adapt`, de modo a lidar com os *pontos de adaptação* específicos da aplicação. Exemplos de tais políticas são apresentados a seguir.

6.3.4 Políticas-Exemplo

Como prova de conceito para a subsequente avaliação experimental, criamos um conjunto de políticas de particionamento para as fontes de ineficiência discutidas na caracterização. Neste momento, não estamos preocupados com os detalhes de implementação das políticas no sistema mas sim com a suas semânticas e propósitos. Para o leitor interessado, todas as implementações foram incluídas no apêndice B. Além disso, é importante notar que as políticas descritas a seguir não são extensivas, apesar de suficientes para demonstrar a aplicabilidade da ferramenta.

Tarefas vazias (ET, algoritmo 2) Se existirem tarefas que não processam nenhum dado, descontamos as mesmas do número de partições original. A aplicação dessa política resulta na atualização do número de partições no respectivo *ponto de adaptação* (`UNPAction`).

Algoritmo 2 Política que identifica e elimina tarefas vazias de uma execução

```
1: Função: OPT-EMPTY-TASKS( rdd, stages )
2:     repr ← FILTER(stages, "mais recente")
3:     numEmptyTasks ← FILTER(repr.tasks, task.input == 0).size
4:     if numEmptyTasks > 0 then
5:         return UNPACTION(rdd.name, repr.tasks.size − numEmptyTasks)
6:     else
7:         return NOACTION(rdd.name)
```

Spill em memória/disco (SP, algoritmo 3) Se os logs indicarem que as tarefas estão realizando considerável *spill* em memória e/ou disco, pode ser um indicador que a carga atribuída a cada tarefa pode estar sendo excessiva. Tentamos mitigar esse problema modificando o número de partições de acordo com um fator da quantidade de *spill* (bytes) observado no estágio em questão. Como a situação ideal acontece quando nenhum *spill* é observado, aplicar essa política significa aumentar o número de partições com o objetivo de distribuir cargas menores para cada tarefa.

Algoritmo 3 Política que aumenta o número de partições de acordo com o *spill* observado

```
1:  Função: OPT-SPILL( rdd, stages )
2:      repr ← FILTER(stages, "mais recente")
3:      bytesSpilled ← repr.bytesSpilled
4:      shuffleWriteBytes ← repr.shuffleWriteBytes
5:      if shuffleWriteBytes > 0 then
6:          factor ← bytesSpilled / shuffleWriteBytes
7:          if factor > 0 then
8:              numPartitions ← repr.numTasks+ CEIL(factor * repr.numTasks)
9:              return UNPACTION(rdd.name, numPartitions)
10:         else
11:             return NOACTION(rdd.name)
12:     else
13:         return NOACTION(rdd.name)
```

Coleta de lixo (GC, algoritmo 4) Nesse caso, primeiramente utilizamos uma métrica de *skewness* para determinar se coleta de lixo deve ser considerada uma fonte de ineficiência ou não. Se isso o *skewness* estiver acima de um limiar pré-estabelecido, aplicamos uma heurística simples que parte cada tarefa proporcionalmente à mediana do *overhead* de coleta de lixo observado. No contexto, *overhead* é a fração de tempo da tarefa atribuída à coleta de lixo. O resultado dessa política também é uma ação que atualiza o número de partições do *ponto de adaptação* (`UNPAction`).

Desbalanceamento entre tarefas (TI, algoritmo 5) Essa política automatiza a metodologia de análise para balancemento de carga discutido na seção 5.3.4. Classificamos fontes de desbalanceamento como sendo: *(a)* inerente; *(b)* atribuído à distribuição de chaves; *(c)* atribuído a custos variáveis para uma mesma chave. Como discutido anteriormente, existem fontes de desbalanceamento que não conseguem ser mitigadas de forma automática. Por esse motivo, para a maioria de fontes de balancemento encontradas, retornamos ações de aviso apenas (`WarnAction`), exceto para desbalanceamento resultante da distribuição de chaves, que pode ser mitigado com a atualização do particionador para *rangePartitioner* (`UPAction`).

Algoritmo 4 Política que distribui melhor a carga das tarefas ao observar que coleta de lixo é mais predominante em um subconjunto de tarefas (*skewness*)

```
1: Função: OPT-GC( rdd, stages )
2:     repr ← FILTER(stages, "mais recente")
3:     gcOverheads ← repr.taskGcOverheads
4:     sk ← SKEWNESS(gcOverheads)
5:     if HIGHSKEWNESS(sk) then
6:         target ← MEDIAN(gcOverheads)
7:         normalized ← NORMALIZE(gcOverheads, target)
8:         numPartitions ← SUM(normalized)
9:         return UNPACTION(rdd.name, numPartitions)
10:    else
11:        return NOACTION(rdd.name)
```

Algoritmo 5 Política que identifica, caracteriza e cria estratégias (ou avisos) para mitigar o desbalanceamento entre tarefas

```
1: Função: OPT-TASK-IMBALANCE( rdd, stages )
2:     repr ← FILTER(stages, "mais recente")
3:     runTimes ← repr.taskRunTimes
4:     sk ← SKEWNESS(runTimes)
5:     if not HIGHSKEWNESS(sk) then
6:         return NOACTION(rdd.name)
7:     corr1 ← CORRELATION(runTimes, repr.taskShuffleReadBytes)
8:     corr2 ← CORRELATION(runTimes, repr.taskShuffleReadRecords)
9:     HighCorr1 ← HIGHCORRELATION(corr1)
10:    HighCorr2 ← HIGHCORRELATION(corr2)
11:    if HighCorr1 and HighCorr2 then
12:        return UPACTION(rdd.name, rangePartitioner)
13:    else if not HighCorr1 and not HighCorr2 then
14:        return WARNACTION(rdd.name, Inherent)
15:    else
16:        return WARNACTION(rdd.name, VariableCost)
```

6.4 Avaliação Experimental

Nossos testes revisitam os experimentos realizados na caracterização e consideram as mesmas fontes de ineficiência. Todos os resultados apresentados a partir deste ponto incluem o tempo gasto pelo *Analyzer* para decidir quais ações devem ser tomadas em benefício da aplicação, nos casos referentes a execuções com o auxílio da ferramenta. Em todas as configurações testadas, o *overhead* do *Analyzer* considerando uma implementação não-otimizada não ultrapassou 3 segundos. A seção 6.4.1 apresenta os resultados para o PageRank, que representa aplicações iterativas regulares. A seção 6.4.2 descreve os resultados referentes ao Eclat, representando aplicações iterativas irregulares. Finalmente, na seção 6.4.3 discutimos alguns resultados referentes ao Twidd e como a ferramenta se comporta nos casos em que o espectro de ações está limitado pelas capacidades de configuração do ambiente de execução. Com o objetivo de demonstrar a efetividade da ferramenta, retomamos os mesmos resultados apresentados e discutidos no capítulo 5. As categorias indicadas como *Conservativo* e *Adaptativo.Manual* representam esses resultados já vistos. O ambiente utilizado para a avaliação experimental da ferramenta foi a mesma utilizada para caracterização (seção 5.1).

6.4.1 PageRank

Nós avaliamos o PageRank em diversos níveis de paralelismo no particionamento de entrada, que determina como os nós do grafo são distribuídos entre as máquinas do *cluster* e, consequentemente, como o padrão de comunicação entre as arestas do grafo se dará em cada iteração do algoritmo. A figura 6.7 apresenta os resultados encontrados, em três cenários diferentes: conservativo, adaptativo manual e adaptativo obtido pela ferramenta.

O cenário conservativo se refere ao comportamento padrão do sistema quando apenas variamos a quantidade de partições de entrada, valor que é reutilizado para as etapas de redução a toda iteração. Nos referimos a esse comportamento como *Conservativo* porque o número de blocos do HDFS do arquivo de entrada tem impacto direto no paralelismo de toda a execução da aplicação.

O modo *Adaptivo.Manual* tem esse nome por tomar vantagem do melhor paralelismo observado na versão conservativa. Por esse motivo, tem a característica de ser um conjunto de testes extensivos que demonstra, acima de tudo, a oportunidade para ganhos em desempenho contra a versão conservativa.

Por último, apresentamos os resultados da abordagem adaptativa obtida pela ferramenta de otimização descrita neste capítulo (*Adaptativo.Ferramenta*). Utilizamos

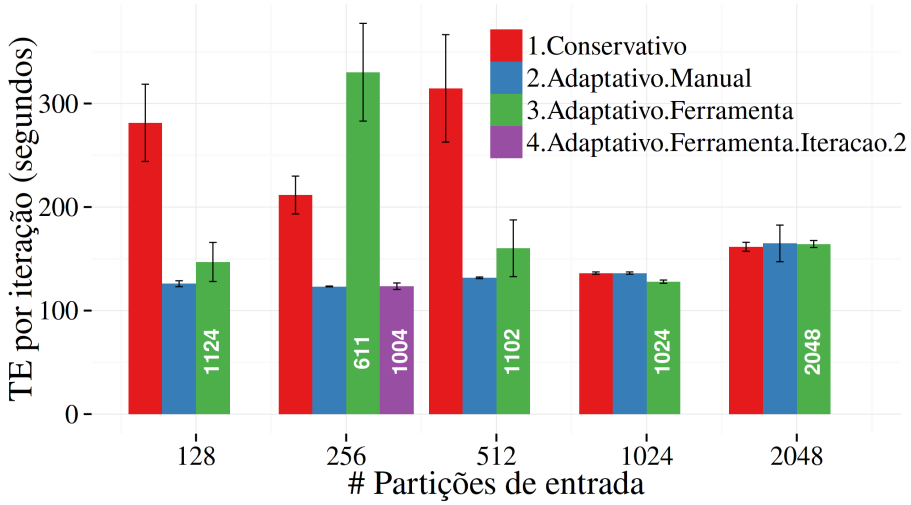


Figura 6.7: O PageRank é composto por estágios do tipo P_{eq}. Os resultados obtidos pela ferramenta não perdem, em geral, quando comparados com a solução manual, que requer mais execuções. Um comportamento interessante é destacado no ponto com 256 partições, onde a ferramenta é usada de forma iterativa, refinando a solução ao atacar diferentes fontes de ineficiência.

os logs das execuções conservativas (*Conservativo*) como entrada para as execuções da ferramenta.

Como visto na seção 5.3.3, com o *Adaptativo.Manual* conseguimos melhorar o desempenho geral. Além disso, os resultados obtidos com *Adaptativo.Ferramenta* foram capazes de acompanhar o melhor caso de adaptação manual (*Adaptativo.Manual*), que mostra como o *framework* aplicado juntamente com as políticas de exemplo foram eficazes em otimizar o programa e com a vantagem de ser uma abordagem automática.

A execução com 256 partições mostra um comportamento interessante relativo ao funcionamento da ferramenta. Em sua implementação atual, a ferramenta aplica apenas uma ação, definida por uma única política, em cada ponto de adaptação e a cada execução da aplicação. Em alguns casos, a primeira rodada de reconfigurações pode implicar na exposição de um segundo gargalo de desempenho. De fato, o resultado para 256 partições na primeira execução de *Adaptativo.Ferramenta* foi pior do que o observado em *Conservativo*. O número de partições escolhido pela ferramenta ao aplicarmos a ação `UNPAction` para 256 partições (611, indicado em branco) ficou próximo ao resultado conservativo para 512 partições. Mais especificamente, a ação retornada pelo *Analyzer* foi aumentar o número de partições baseado na elevada quantidade de *spill* das tarefas, uma ação *UNPAction* baseada na política **SP** (seção 6.1). Após resolver o problema relacionado ao *spill*, a nova execução passou a sofrer com um elevado *overhead* na coleta de lixo, caracterizado por algumas tarefas demandando excessivo

tempo para término (*stragglers*). Uma segunda rodada de execução utilizando a ferramenta e o novo log obtido foi suficiente para atingir o desempenho desejado, ou seja, próximo ao observado em *Adaptativo.Manual*. Neste último caso, o *Analyzer* retornou para o ponto de adaptação em questão uma ação *UNPAction* relativa à política **GC** apresentada na seção 6.1. Neste estudo de caso em especial, o resultado é destacado como *Adaptativo.Ferramenta.Iteracao.2*.

6.4.2 Eclat

Avaliamos o algoritmo Eclat variando o nível de paralelismo do *ponto de adaptação* indicado na figura 4.3, ou seja, na etapa onde as contagens locais são agregagas globalmente entre os executores do *cluster*. A figura 6.8 apresenta os resultados obtidos nos mesmos três cenários considerados para o PageRank na seção 6.4.1. Como antes, *Conservativo* indica que a etapa de redução foi configurada com o mesmo número de partições de entrada em toda iteração. O identificador *Adaptativo.Manual* considera a heurística baseada na evolução da quantidade de dados processados ao longo das iterações, descrita na seção 5.3.3. Por fim, *Adaptativo.Ferramenta* representa as execuções adaptadas pela ferramenta utilizando os logs das execuções conservativas como entrada.

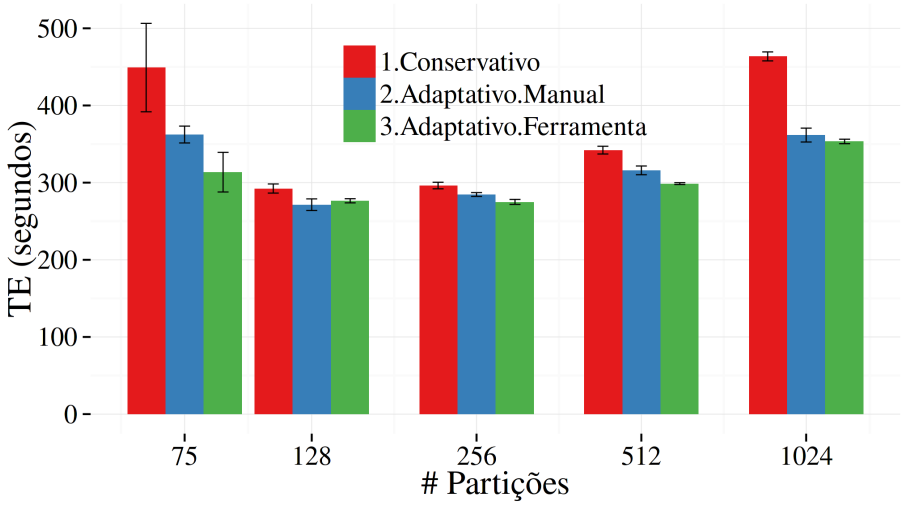


Figura 6.8: O Eclat é composto por estágios do tipo P_{sim}. Os resultados obtidos pela ferramenta são equivalentes aos de adaptação manual, tanto para a métrica de tempo de execução quanto na redução do número de tarefas submetidas.

Nas execuções do grupo *Adaptativo.Ferramenta*, a política predominante nesse caso é a que lida com as tarefas vazias **ET**. Como ela não assume nada sobre a natureza da aplicação alvo, é ideal para capturar o padrão de comportamento típico de problemas

de mineração de padrões frequentes, que tendem a crescer muito inicialmente e encolher expressivamente no final. De fato, o objetivo esperado é que a reconfiguração seja capaz de lidar com o comportamento real das aplicações, que pode ser regular (figura 5.8) ou não (figura 5.9).

Ambas soluções adaptativas são especialmente eficientes em lidar com casos em que o número de partições ideal foi subestimado ou superestimado (75 e 1024 partições de entrada). O ganho é pequeno em configurações intermediárias à medida que se aproxima do paralelismo ótimo, no qual a geração de candidatos (fase de mapeamento) supera a etapa de agregação dos contadores globalmente (etapa de redução). Além disso, todo resultado obtido pela ferramenta apresenta pequenos ganhos ou pelo menos comportamente equivalente ao desempenho observado pela execução adaptativa manual.

As execuções adaptativas também demonstram a capacidade de redução no número de tarefas necessárias para realizar o mesmo trabalho, reduzindo assim o *overhead* de disparo das mesmas. Indicamos esse tipo de ganho dentro das barras na forma de porcentagem de redução quando comparado com a abordagem conservativa (figura 6.9). O número de tarefas disparadas é reduzido em, pelo menos, 78,4% para *Adaptativo.Manual* e por, pelo menos, 73,6% para *Adaptativo.Ferramenta*, mesmo quando os ganhos em tempo de execução são modestos.

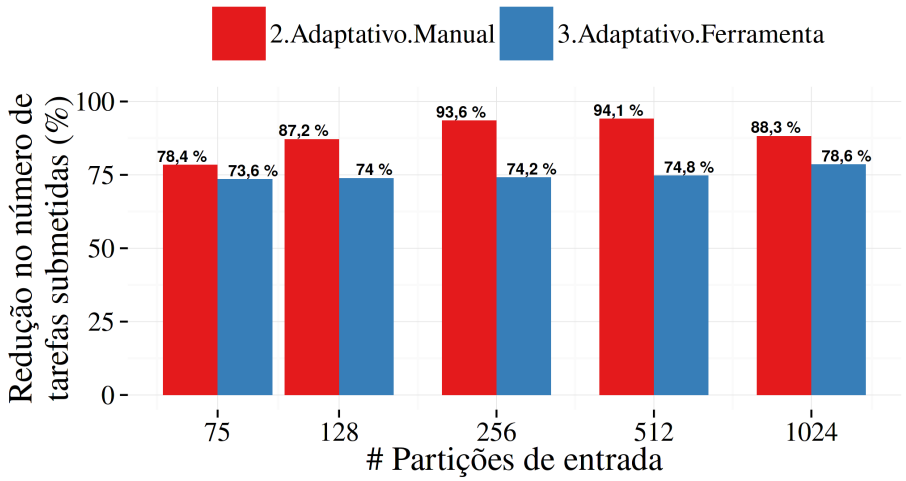


Figura 6.9: O Eclat é composto por estágios do tipo P_{sim}. Porcentagens indicam redução na utilização de recursos das abordagens adaptativas em relação à conservativa *(1.Convervativo)*, indicada na figura 6.8.

6.4.3 Twidd

A figura 6.10 ilustra as mensagens geradas pela ferramenta quando o Twidd é re-executado com um log passado (exemplo discutido na seção 5.3.4). As fontes de desbalanceamento discutidas são capturadas de maneira automática pela ferramenta: no estágio 4, onde as μ-árvores (*muTrees*) são fundidas, o desbalanceamento é devido ao custo de processamento variável entre registros com a mesma chave; no estágio 5, onde as ρ-árvores (*rhoTrees*) são fundidas, a fonte de desbalanceamento é inerente à natureza do algoritmo. Como a capacidade atual da máquina de execução do sistema não permite que esses problemas sejam contornados sem a intervenção do usuário, as ações são meramente avisos apresentados para o usuário durante a execução da aplicação.

```
WARN OptHelper:  WarnAction(muTrees, VariableCost) (task-imbalance,257)
WARN OptHelper:  WarnAction(rhoTrees, Inherent) (task-imbalance,257)
```

Figura 6.10: Ações do tipo *Warn* produzem avisos para o usuário

Capítulo 7

Conclusão

Entender elementos que afetam o desempenho de aplicações massivamente paralelas é uma tarefa difícil, entre outros motivos, pelos desafios introduzidos pelos ambientes de processamento. Vimos que a motivação principal para trabalhos nessa linha se apoia no compromisso inevitável entre aumentar o nível de abstração, facilitando a programação de programas paralelos, e sacrificar a simplicidade das soluções, com o objetivo de obter melhor desempenho. Através da caracterização de aplicações típicas, foi possível observar um espectro intermediário entre esses dois extremos, em que demonstramos oportunidades para melhorias no desempenho sem sacrificar nenhum aspecto do modelo de execução. Com o objetivo de amadurecer o entendimento das fontes de ineficiência das aplicações e definir ações simples para otimizar o desempenho, adotamos uma metodologia baseada em três passos.

Primeiro definimos um conjunto de dimensões e aplicações típicas de cenários de análise de dados e processamento massivo. Esse aspecto foi importante principalmente no sentido de reduzir a complexidade dos diversos fatores que influenciam o desempenho no modelo de execução considerado. Em particular, ao diagnosticar fontes de ineficiência sob perspectivas e dimensões específicas, fomos capazes de tirar conclusões confiáveis através de escopos de análise mais restritos.

Um segundo passo consistiu de uma avaliação experimental que possibilitou associar classes de aplicações, padrões de comportamento e oportunidades. Ao identificar oportunidades para ajuste de desempenho levando em conta a categoria e padrão de comunicação das aplicações, ganhamos em generalidade e flexibilidade para aplicar as mesmas metodologias em cenários similares. A saída desse passo foi um conjunto de lições aprendidas sobre o desempenho em processamento massivo e principalmente, o domínio de ações possíveis para o ajuste da eficiência das aplicações.

O ferramental obtido nas etapas anteriores resultou, em um terceiro momento,

no projeto e na avaliação de uma ferramenta que automatiza o processo de ajuste de desempenho em aplicações recorrentes. A ferramenta se baseia em políticas extensíveis para ajustar execuções futuras de uma mesma aplicação e para isso considera os padrões de comunicação das categorias de aplicações consideradas. A estratégia proposta se mostrou eficaz em obter resultados equivalentes à reconfiguração manual, com a vantagem de ser um processo automatizado e demandar menos execuções anteriores.

Trabalhos futuros

Consideramos duas linhas principais para contribuições futuras. A primeira delas tem o objetivo de explorar mais a fundo as capacidades do *framework* desenvolvido neste trabalho e, para isso, foca na criação de modelos mais sofisticados, ainda no contexto de aplicações recorrentes, de forma a considerar várias fontes de ineficiência de uma vez. Acreditamos que as alterações na implementação do *framework* seriam mínimas e os modelos poderiam ser escritos na forma de políticas e ações, da mesma forma que acontece na versão atual. Além disso, seria interessante investigar a possibilidade de integrar ajustes de desempenho que modifiquem a topologia dos DAGs de execução. Por um lado, estratégias como essa são mais complexas e demandam um cuidado especial com a integridade da semântica do código do usuário entre ajustes. Por outro lado, mudar a topologia dos DAGs de execução pode trazer ganhos consideráveis, visto que poderíamos aplicar estratégias empíricas para escolha do melhor conjunto de operadores dada uma carga de trabalho. Ainda assim, as considerações feitas neste trabalho sobre os padrões de comunicação das aplicações e as abordagens especializadas para o ajuste de desempenho em cada caso seriam importantes em um contexto de modificação do DAG de execução, pois essas conclusões poderiam auxiliar na definição de custos de operadores do sistema e UDFs (funções definidas pelo usuário), etapa primordial na maioria das estratégias de definição e avaliação de planos de execução alternativos.

A segunda linha tem o objetivo de lidar com gargalos de desempenho em tempo de execução, sem nenhuma suposição sobre a natureza da aplicação. Esse aspecto é desafiador pois implica em modificações da máquina de execução do *framework* (Spark, por exemplo) e também precisa ser feito de forma a não sacrificar a transparência oferecida em programação ao usuário. Um requisito importante nesse contexto é que qualquer esforço no sentido de aprimorar a capacidade de otimização de programas em tempo de execução precisa utilizar e interpretar diversas instrumentações do ambiente e saber interpretá-las de forma consistente. Em particular, a capacidade de caracterizar o tempo de vida da tarefa através de métricas mais precisas pode possibilitar

que o escalonador se beneficie de uma sobreposição de recursos (memória, disco e processamento), melhorando o desempenho como um todo. Mais uma vez, entretanto, a complexidade da questão causa/efeito no diagnóstico de desempenho se torna a maior dificuldade.

Referências Bibliográficas

Agarwal, S.; Kandula, S.; Bruno, N.; Wu, M.-C.; Stoica, I. & Zhou, J. (2012). Re-optimizing data-parallel computing. Em *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pp. 21–21, Berkeley, CA, USA. USENIX Association.

Bruno, N.; Agarwal, S.; Kandula, S.; Shi, B.; Wu, M.-C. & Zhou, J. (2012). Recurring job optimization in scope. Em *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pp. 805–806, New York, NY, USA. ACM.

Chaiken, R.; Jenkins, B.; Larson, P.-A.; Ramsey, B.; Shakib, D.; Weaver, S. & Zhou, J. (2008). Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276.

Dean, J. & Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Ghodsi, A.; Zaharia, M.; Hindman, B.; Konwinski, A.; Shenker, S. & Stoica, I. (2011). Dominant resource fairness: Fair allocation of multiple resource types. Em *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pp. 323–336, Berkeley, CA, USA. USENIX Association.

Gog, I.; Giceva, J.; Schwarzkopf, M.; Vaswani, K.; Vytiniotis, D.; Ramalingam, G.; Costa, M.; Murray, D. G.; Hand, S. & Isard, M. (2015). Broom: Sweeping out garbage collection from big data systems. Em *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland. USENIX Association.

Han, J.; Pei, J. & Yin, Y. (2000). Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12.

Herodotou, H. & Babu, S. (2011). Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. Em *PVLDB: Proceedings of the VLDB Endowment*, volume 4, pp. 1111–1122.

Herodotou, H.; Lim, H.; Luo, G.; Borisov, N.; Dong, L.; Cetin, F. B. & Babu, S. (2011). Starfish: A self-tuning system for big data analytics. Em *CIDR*, pp. 261–272.

Hindman, B.; Konwinski, A.; Zaharia, M.; Ghodsi, A.; Joseph, A. D.; Katz, R.; Shenker, S. & Stoica, I. (2011). Mesos: A platform for fine-grained resource sharing in the data center. Em *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pp. 295–308, Berkeley, CA, USA. USENIX Association.

Isard, M.; Budiu, M.; Yu, Y.; Birrell, A. & Fetterly, D. (2007). Dryad: Distributed data-parallel programs from sequential building blocks. Em *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pp. 59–72, New York, NY, USA. ACM.

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley professional computing. Wiley.

Ke, Q.; Isard, M. & Yu, Y. (2013). Optimus: A dynamic rewriting framework for data-parallel execution plans. Em *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pp. 15–28, New York, NY, USA. ACM.

Ke, Q.; Prabhakaran, V.; Xie, Y.; Yu, Y.; Wu, J. & Yang, J. (2011). Optimizing data partitioning for data-parallel computing. Em *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pp. 13–13, Berkeley, CA, USA. USENIX Association.

Maas, M.; Harris, T.; Asanović, K. & Kubiatowicz, J. (2015). Trash day: Coordinating garbage collection in distributed systems. Em *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland. USENIX Association.

Magno, G.; Comarela, G.; Saez-Trumper, D.; Cha, M. & Almeida, V. (2012). New kid on the block: Exploring the google+ social graph. Em *Proceedings of the 2012 ACM conference on Internet measurement conference*, IMC '12, pp. 159–170, New York, NY, USA. ACM.

Malewicz, G.; Austern, M. H.; Bik, A. J.; Dehnert, J. C.; Horn, I.; Leiser, N. & Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. Em *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pp. 135–146, New York, NY, USA. ACM.

Nguyen, K.; Wang, K.; Bu, Y.; Fang, L.; Hu, J. & Xu, G. (2015). Facade: A compiler and runtime for (almost) object-bounded big data applications. Em *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pp. 675–690, New York, NY, USA. ACM.

Ousterhout, K.; Panda, A.; Rosen, J.; Venkataraman, S.; Xin, R.; Ratnasamy, S.; Shenker, S. & Stoica, I. (2013). The case for tiny tasks in compute clusters. Em *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, Berkeley, CA. USENIX.

Ousterhout, K.; Rasti, R.; Ratnasamy, S.; Shenker, S. & Chun, B. G. (2015). Making sense of performance in data analytics frameworks. Em *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 293–307, Oakland, CA. USENIX Association.

Page, L.; Brin, S.; Motwani, R. & Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab. Previous number = SIDL-WP-1999-0120.

Shi, J.; Qiu, Y.; Minhas, U. F.; Jiao, L.; Wang, C.; Reinwald, B. & Özcan, F. (2015). Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow.*, 8(13):2110–2121.

Shvachko, K.; Kuang, H.; Radia, S. & Chansler, R. (2010). The hadoop distributed file system. Em *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pp. 1–10, Washington, DC, USA. IEEE Computer Society.

Vavilapalli, V. K.; Murthy, A. C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; Saha, B.; Curino, C.; O'Malley, O.; Radia, S.; Reed, B. & Baldeschwieler, E. (2013). Apache hadoop yarn: Yet another resource negotiator. Em *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pp. 5:1–5:16, New York, NY, USA. ACM.

Veloso, A.; Meira, Jr., W.; Ferreira, R.; Guedes, D. & Parthasarathy, S. (2004). Asynchronous and anticipatory filter-stream based parallel algorithm for frequent itemset mining. Em *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, PKDD '04, pp. 422–433, New York, NY, USA. Springer-Verlag New York, Inc.

Xin, R. (2015). Spark-7075: Project tungsten. https://issues.apache.org/jira/browse/SPARK-7075. Visitado em 31/10/2016.

Zaharia, M. (2015). Spark-9850: Adaptive execution in spark. https://issues.apache.org/jira/browse/SPARK-9850. Visitado em 31/10/2016.

Zaharia, M.; Borthakur, D.; Sen Sarma, J.; Elmeleegy, K.; Shenker, S. & Stoica, I. (2010). Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. Em *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pp. 265–278, New York, NY, USA. ACM.

Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M. J.; Shenker, S. & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Em *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pp. 2–2, Berkeley, CA, USA. USENIX Association.

Apêndice A

Premissas do Projeto Fatorial

Aplicamos a metodologia do projeto fatorial $2^k r$ para estimar o impacto de alguns fatores relevantes no contexto do trabalho. Os resultados obtidos são apresentados e discutidos na seção 5.3.1. O objetivo dessa seção é discutir a qualidade de tais modelos. O projeto fatorial modela bem um sistema quando as medições obtidas satisfazem seguintes premissas [Jain, 1991]:

1. Efeitos dos fatores são aditivos;
2. Erros são aditivos;
3. Erros são independentes dos níveis do fator;
4. Erros são normalmente distribuídos;
5. Erros têm a mesma variância para todos os níveis do fator.

Como muitas dessas premissas são relacionadas, em geral é comum que um sistema satisfaça todas elas ou nenhuma. Assim, as mais importantes e consideradas neste trabalho são: (4) normalidade de erros e (5) variância uniforme para os níveis dos fatores. Para realizar essa verificação, duas análises visuais foram realizadas. A primeira consiste de um gráfico Quantil-Quantil dos erros para cada métrica, que indicam os desvios das replicações sobre a média em cada configuração do projeto. Nesse tipo de gráfico, os quantis de uma distribuição (erros medidos) são plotados contra os quantis de outra distribuição (nesse caso, uma normal). Uma reta nesse gráfico, indica a proximidade dos erros com uma distribuição normal. A segunda consiste de um gráfico de desvios sobre a média (ou seja, os erros) em função das respostas obervadas (média de cada configuração). Um bom resultado nesse tipo de gráfico é identificado por uma nuvem de pontos, sem tendências, uniformemente espalhados sobre o eixo y.

Caso essas premissas não se verifiquem, uma transformação sobre as medições se faz necessária, com o objetivo de eliminar as tendências e aproximar o modelo à normalidade de forma satisfatória. Transformações foram necessárias para a maioria dos modelos discutidos. Por isso, é importante mencionar que as conclusões sobre os efeitos de cada fator e suas interações são feitas sobre uma função das variáveis resposta, isto é, sobre os resultados transformados. Entretanto, como as funções de transformações são crescentes, conclusões referentes ao comportamento das métricas podem ainda ser feitas, o que valida a discussão realizada na seção 5.3.1.

A.1 Qualidade dos Modelos

Ao todo, realizamos 8 projetos fatoriais, refletindo as combinações de 2 algoritmos (Twidd e Eclat), em duas configurações de *cluster* (5 e 9 executores) e considerando duas métricas (tempo de execução `TE` e *overhead* em coleta de lixo `GC`).

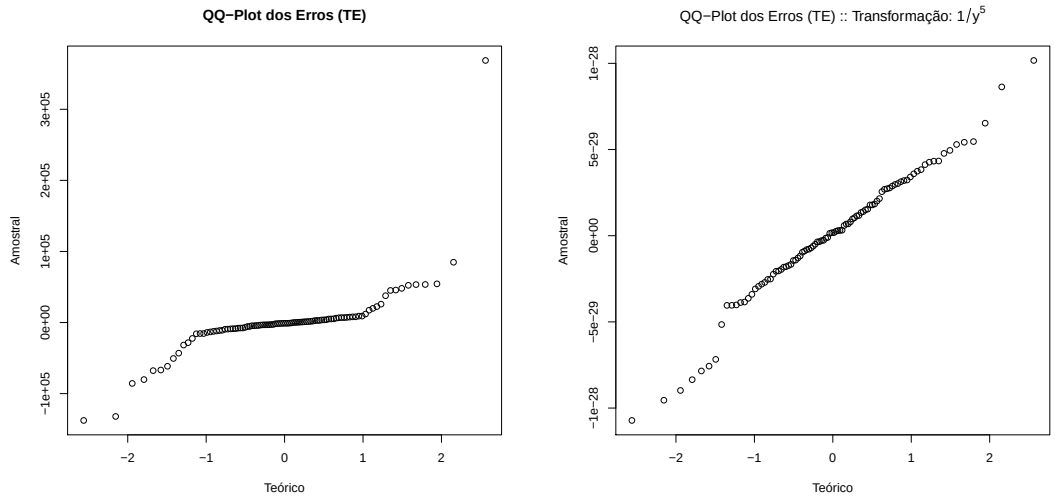
Em linhas gerais, os projetos envolvendo menos executores apresentaram uma qualidade melhor, mesmo depois das transformações aplicadas. Uma justificativa plausível para essa observação é que à medida que adicionamos mais máquinas no *cluster*, maior as possibilidades de interações entre os elementos do sistema e portanto, maior a sua complexidade. Apesar disso, a maioria das transformações aplicadas foram capazes de aproximar o modelo à normalidade. As seções seguintes apresentam os gráficos das análises visuais para cada projeto. A nomenclatura utilizada para identificar o projeto é a seguinte: `<algoritmo>-<numero_de_executores>-<métrica>`.

`Twidd-5-TE`

As figuras A.1a e A.1b ilustram os gráficos que verificam a normalidade dos erros antes e depois da transformação aplicada. Nesse caso, utilizamos uma transformação $\frac{1}{y^5}$. Observe como a transformação foi capaz de lidar com o achatamento dos valores intermediários, resultando em uma curva mais próxima a uma reta. Perceba também o espalhamento mais uniforme dos desvios (figuras A.2a e A.2b). O valor do expoente foi selecionado empiricamente.

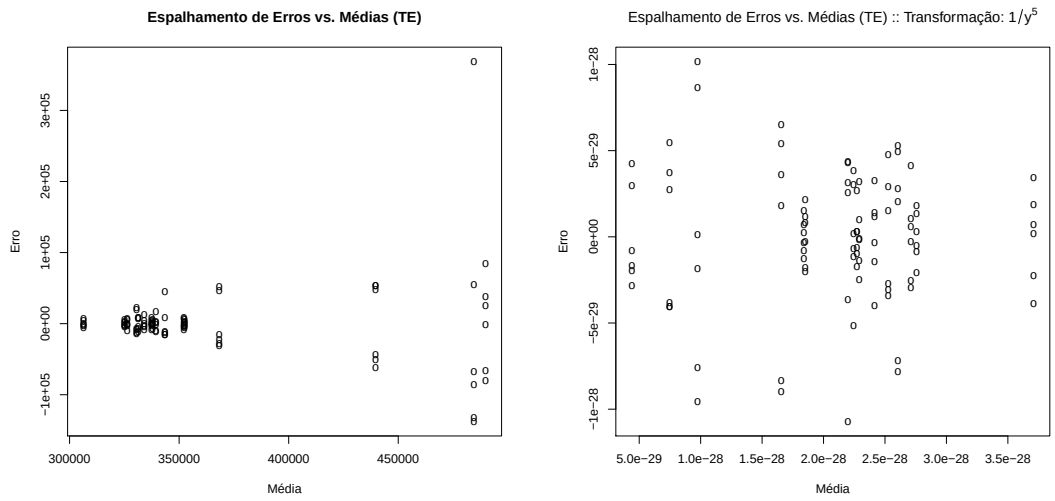
`Twidd-5-GC`

Mais uma vez, uma transformação de potência foi aplicada e dessa vez com o objetivo de suavizar erros discrepantes (*outliers*). Por se tratar de uma métrica de proporção (entre 0 e 1), o expoente escolhido foi positivo. As figuras A.3a e A.3b representam



(a) Sem transformação (b) Transformação $\frac{1}{x^5}$

Figura A.1: Twidd-5-TE: Gráfico Quantil-Quantil dos Erros vs. Normal



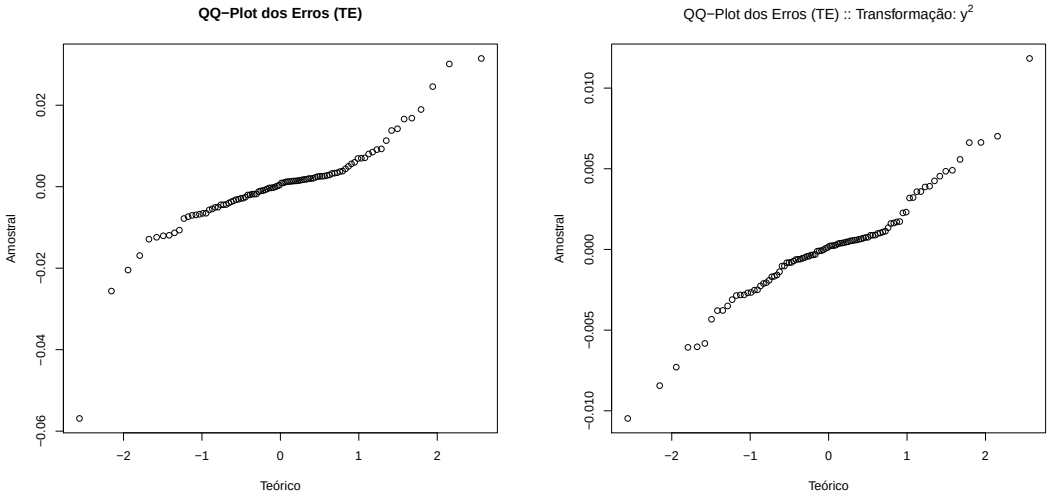
(a) Sem transformação (b) Transformação $\frac{1}{x^5}$

Figura A.2: Twidd-5-TE: Gráficos de espalhamento de erros

os resultados antes e depois da transformação y^2. As figuras A.4a A.4b representam a evolução no quesito espalhamento dos desvios sobre a média.

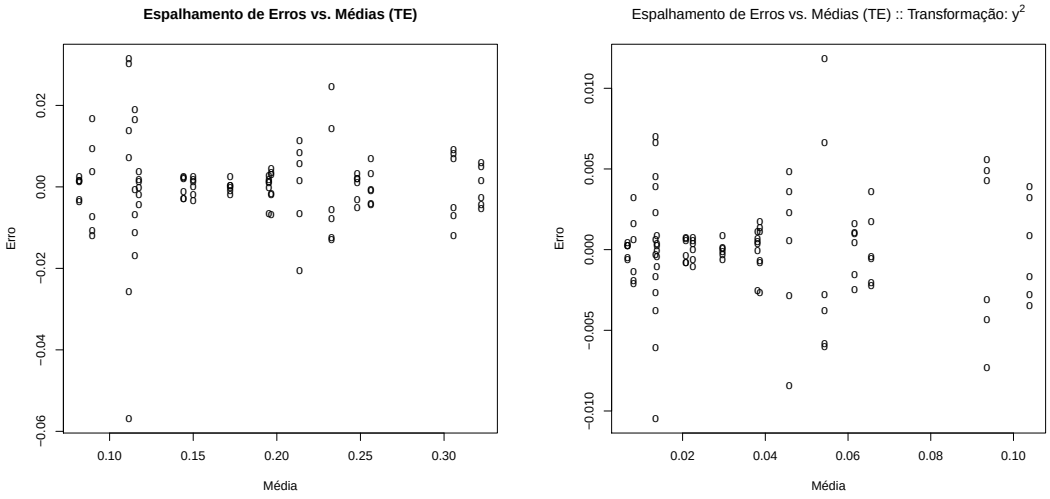
Eclat-5-TE

Esse modelo não necessitou nenhuma transformação pois os erros derivados por si só exibiram um comportamento bem próximo a uma normal (figura A.5a). Além disso, como esperado, o espalhamento dos desvios foi satisfatório (figura A.6a).



(a) Sem transformação (b) Transformação y^2

Figura A.3: Twidd-5-GC: Gráfico Quantil-Quantil dos Erros vs. Normal

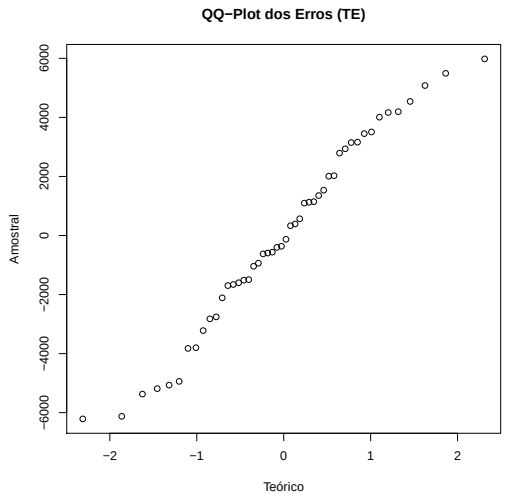


(a) Sem transformação (b) Transformação y^2

Figura A.4: Twidd-5-GC: Gráficos de espalhamento de erros

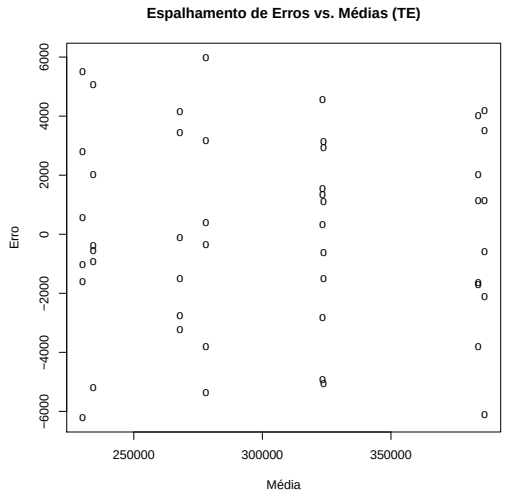
`Eclat-5-GC`

Nesse caso, observamos a mesma tendência do modelo `Twidd-5-TE` e portanto aplicamos uma transformação de potência. Como o achatamento central da curva é mas suave, esperamos que o expoente adequado seja menor. De fato, empiricamente observamos que $\frac{1}{y}$ aproximou a curva de uma reta. As figuras A.7a, A.7b, A.8a e A.8b representam os resultados do teste de normalidade e espalhamento de erros.



(a) Sem transformação

Figura A.5: Eclat-5-TE: Gráfico Quantil-Quantil dos Erros vs. Normal

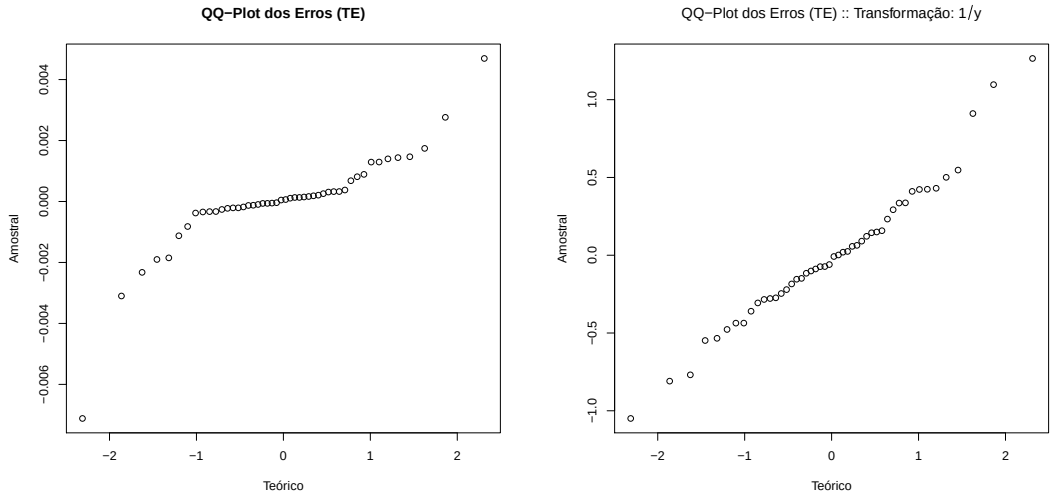


(a) Sem transformação

Figura A.6: Eclat-5-TE: Gráficos de espalhamento de erros

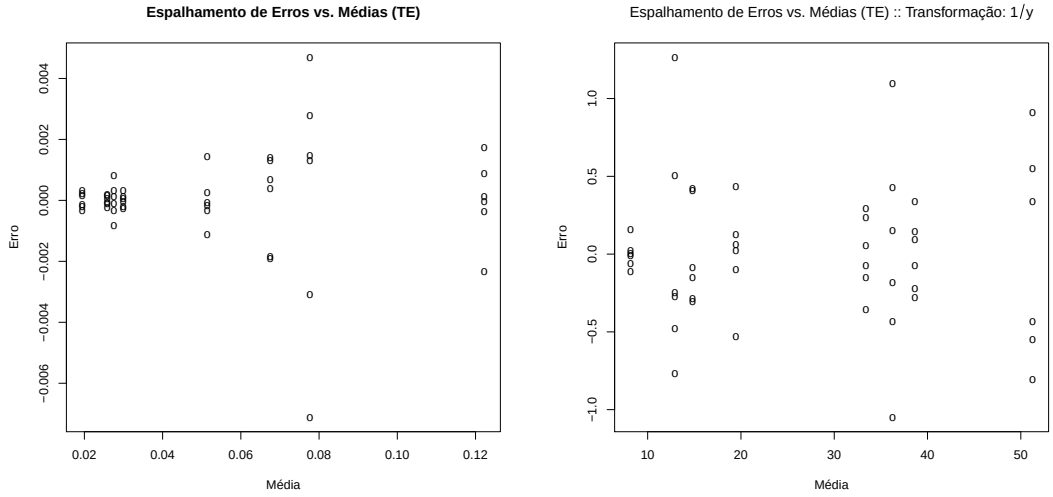
`Twidd-9-TE`

Os erros desse projeto se comportaram de maneira semelhante ao projeto `Twidd-5-TE`, apesar de apresentar maior irregularidade. Assim, aplicamos a mesma transformação $\frac{1}{y^5}$. A melhora foi considerável, tanto no quesito normalidade (figuras A.9a e A.9b) quanto no quesito espalhamento de erros (figuras A.10a A.10b).



(a) Sem transformação (b) Transformação $\frac{1}{y}$

Figura A.7: Eclat-5-GC: Gráfico Quantil-Quantil dos Erros vs. Normal

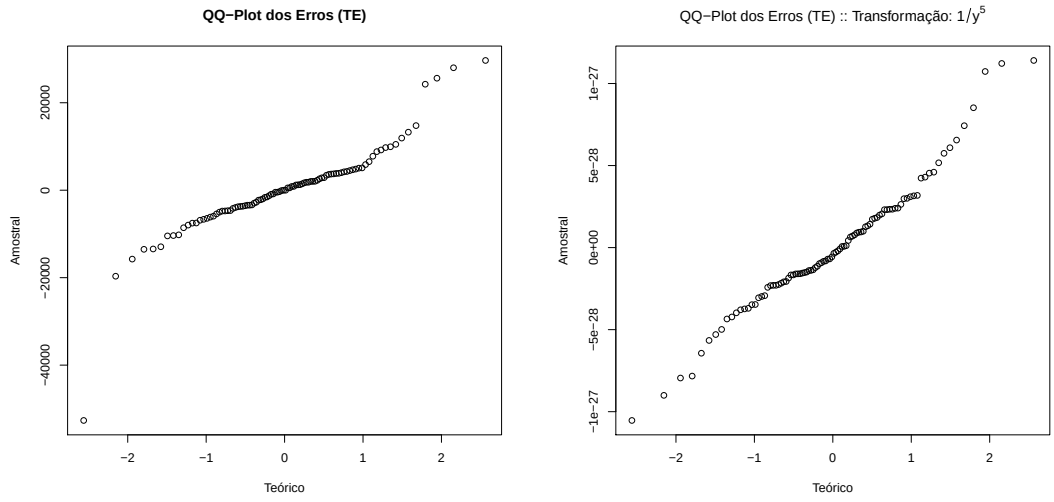


(a) Sem transformação (b) Transformação $\frac{1}{y}$

Figura A.8: Eclat-5-GC: Gráficos de espalhamento de erros

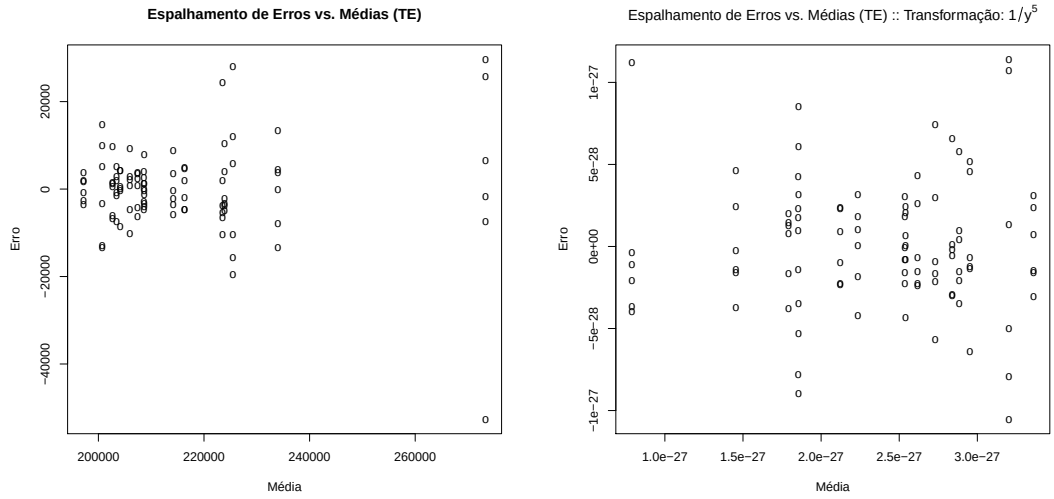
`Twidd-9-GC`

Por se tratar de um conjunto de experimentos especialmente afetado pelo *overhead* com coleta de lixo, o efeito da transformação em normalizar os erros, apesar de positivo, foi pequeno. O melhor resultado encontrado foi através da aplicação da mesma função do seu projeto par `Twidd-5-GC`, ou seja, y^2. O quesito normalidade está ilustrado nas figuras A.11a e A.11b. As figuras A.12a e A.12b representa a evolução em relação



(a) Sem transformação (b) Transformação $\frac{1}{x^5}$

Figura A.9: Twidd-9-TE: Gráfico Quantil-Quantil dos Erros vs. Normal



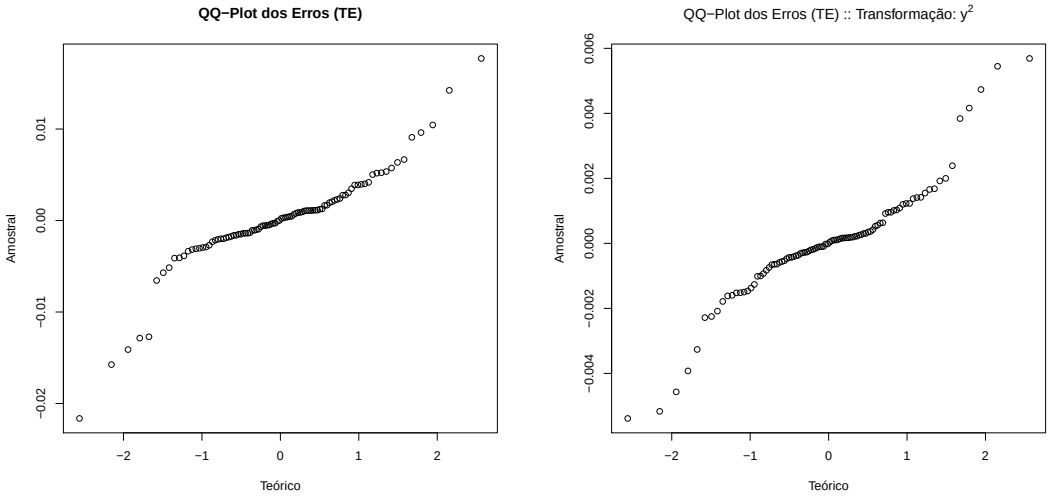
(a) Sem transformação (b) Transformação $\frac{1}{x^5}$

Figura A.10: Twidd-9-TE: Gráficos de espalhamento de erros

ao espalhamento dos erros. Observe que a dificuldade maior da transformação foi em eliminar a tendência inicial do modelo (média entre 0 e 0.15).

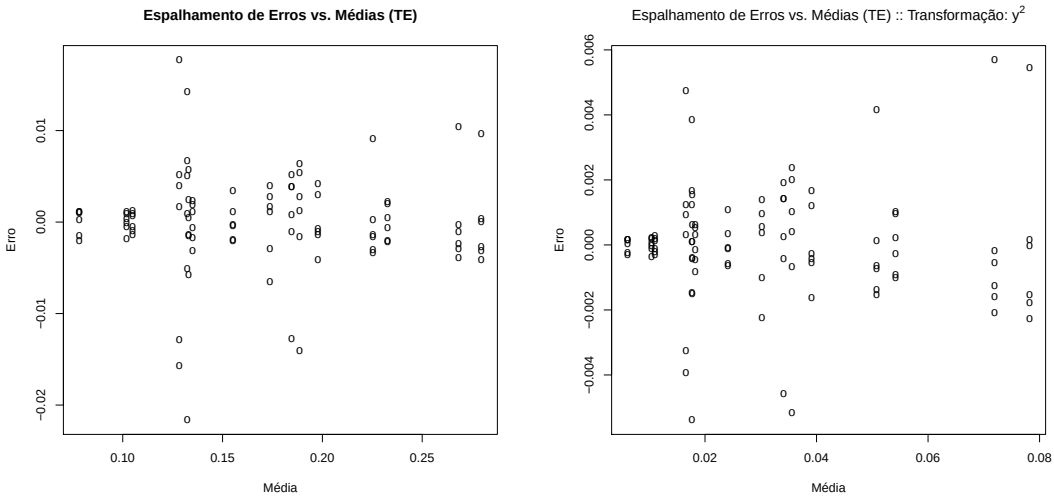
`Eclat-9-TE`

Não encontramos uma transformação que fosse satisfatória em melhorar a normalidade desse modelo.



(a) Sem transformação (b) Transformação y^2

Figura A.11: Twidd-9-GC: Gráfico Quantil-Quantil dos Erros vs. Normal

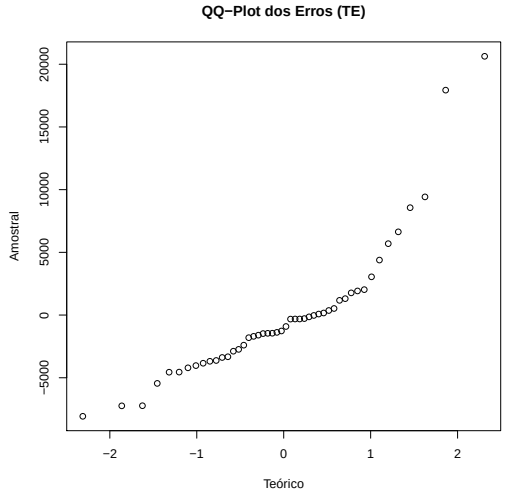


(a) Sem transformação (b) Transformação y^2

Figura A.12: Twidd-9-GC: Gráficos de espalhamento de erros

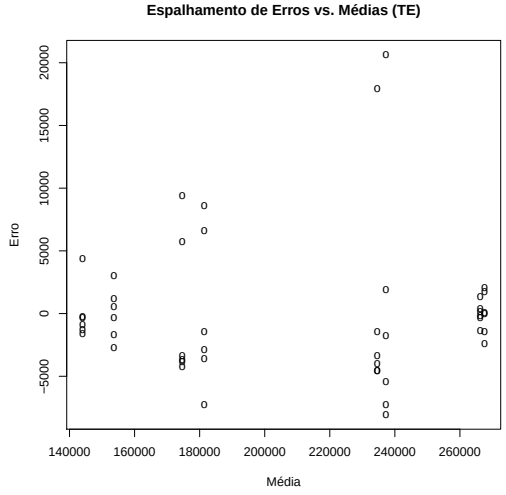
`Eclat-9-GC`

Para esse modelo, aplicamos a transformação $\frac{1}{\sqrt{y}}$ com o objetivo de tratar as tendências de cauda da distribuição. Observe que a transformação consegue consertar a maioria dos ponto exceto um *outlier* superior (figuras A.15a e A.15b). As mesmas considerações podem ser feitas sobre o espalhamento dos erros (figuras A.16a e A.16b).



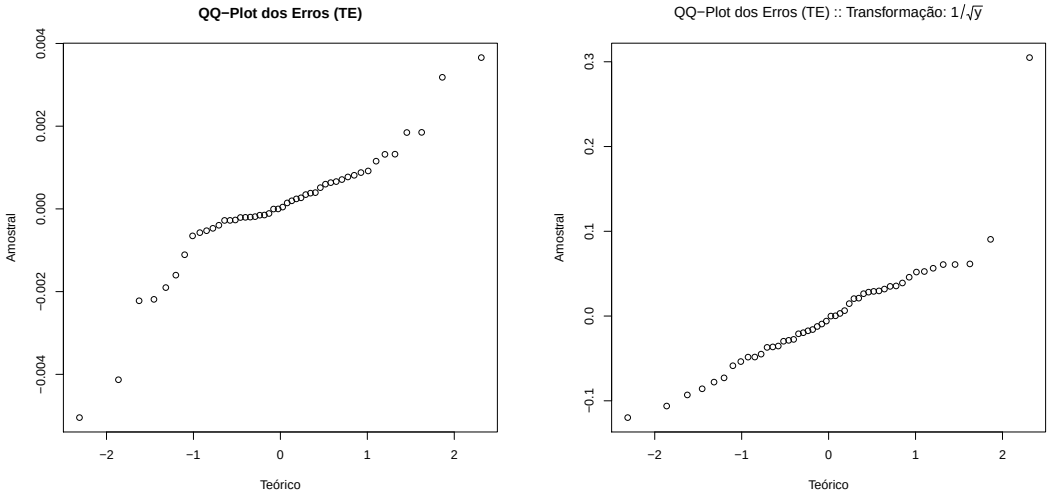
(a) Sem transformação

Figura A.13: Eclat-9-TE: Gráfico Quantil-Quantil dos Erros vs. Normal



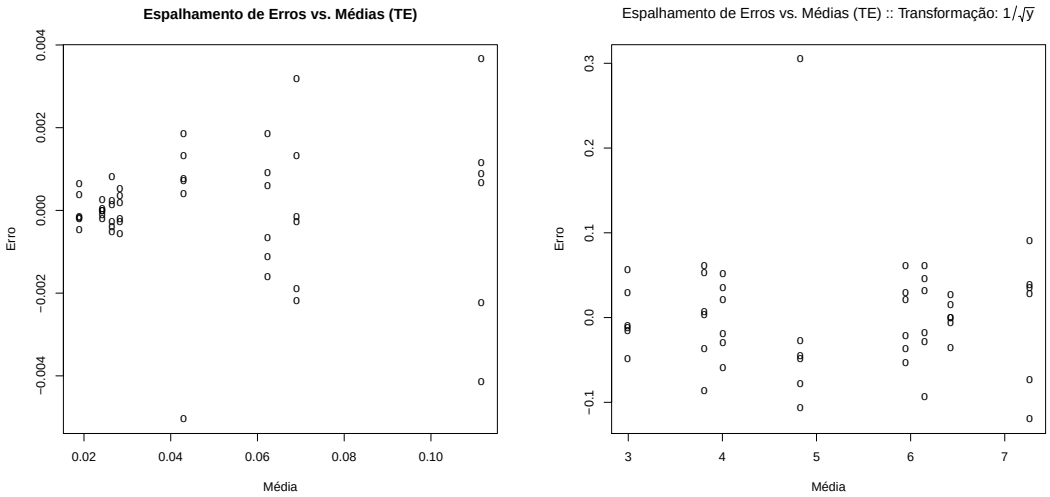
(a) Sem transformação

Figura A.14: Eclat-9-TE: Gráficos de espalhamento dos erros



(a) Sem transformação (b) Transformação

Figura A.15: Eclat-9-GC: Gráfico Quantil-Quantil dos Erros vs. Normal



(a) Sem transformação (b) Transformação

Figura A.16: Eclat-9-GC: Gráficos de espalhamento de erros

Apêndice B

Repositório de códigos

A seguir apresentamos os códigos das políticas discutidas na seção 6.3.3. Essas implementações exemplificam o uso da API de políticas, exportada pela ferramenta de reconfiguração (capítulo 6).

```
1  object EmptyTasksPolicy extends PartitioningPolicy {
2    override def adapt(env:  Environment, stats:  AdaptivePointStats):  Action = {
3      val stages = stats.stages
4      val rdd = stats.rdd
5      logInfo (s"${rdd}:  optimizing for emptyTasks")
6      val repr = getRepr (stages)
7      val emptyTasks = repr.emptyTasks
8      val numEmptyTasks = emptyTasks.size
9      if (numEmptyTasks > 0) {
10       UNPAction (rdd.name, (repr.numTasks - numEmptyTasks).toInt)
11     } else {
12       NOAction (rdd.name)
13     }
14   }
15 }
```

Figura B.1: Política de particionamento para tarefas vazias

```
1  object SpillPolicy extends PartitioningPolicy {
2    override def adapt(env:  Environment, stats:  AdaptivePointStats):  Action = {
3      val stages = stats.stages
4      val rdd = stats.rdd
5
6      logInfo (s"${rdd}:  optimizing for spill")
7      val repr = getRepr (stages)
8      val runTimes = repr.taskRunTimes
9      val bytesSpilled = repr.taskBytesSpilled
10     val shuffleWriteBytes = repr.shuffleWriteBytes
11
12     if (shuffleWriteBytes > 0) {
13       val factor = bytesSpilled / shuffleWriteBytes.toDouble
14       if (factor > 0) {
15         val newNumPartitions = repr.numTasks + math.ceil (factor * repr.numTasks)
16         UNPAction (rdd.name, newNumPartitions.toInt)
17       } else {
18         NOAction (rdd.name)
19       }
20     } else {
21       NOAction (rdd.name)
23     }
24   }
25 }
```

Figura B.2: Política de particionamento para *spill* em memória ou disco

```
1  object GCPolicy extends PartitioningPolicy {
2
3    override def adapt(env:  Environment, stats:  AdaptivePointStats):  Action = {
4      val stages = stats.stages
5      val rdd = stats.rdd
6
7      logInfo (s"${rdd}:  optimizing for GC")
8      val repr = getRepr (stages)
9      val gcOverheads = repr.taskGcOverheads
10     val _skewness = skewness (gcOverheads)
11
12     if (highSkewness (_skewness)) {
13       val target = percentileCalc.evaluate (50)
14       val newNumPartitions = gcOverheads.map (
15           go => math.ceil(go / target).toInt max 1).sum
16       UNPAction (rdd.name, newNumPartitions)
17     } else {
18       NOAction (rdd.name)
19     }
20   }
21 }
```

Figura B.3: Política de particionamento para coleta de lixo com custo muito imprevisível

```
1  object TaskImbalancePolicy extends PartitioningPolicy {
2    /**
3     * The following classes represent the possible causes for task imbalance.
4     * These facts help us to determine the best set of actions to take
5     */
6    private sealed trait SourceOfImbalance
7    private case object NoImbalance extends SourceOfImbalance
8    private case object KeyDist extends SourceOfImbalance
9    private case object Inherent extends SourceOfImbalance
10   private case object VariableCost extends SourceOfImbalance
11
12   private def sourceOfImbalance(stage:  Stage):  SourceOfImbalance = {
13     val runTimes = stage.taskRunTimes
14     val _skewness = skewness (runTimes)
15     logInfo (s"Imbalance skewness ${_skewness}")
16
17     if (!highSkewness(_skewness)) return NoImbalance
18
19     val corr1 = correlation (runTimes, stage.taskShuffleReadBytes)
20     val corr2 = correlation (runTimes, stage.taskShuffleReadRecords)
21     logInfo (s"Correlation between Runtime and ShuffleReadBytes:  ${corr1}")
22     logInfo (s"Correlation between Runtime and ShuffleReadRecords:  ${corr2}")
23     (highCorrelation (corr1), highCorrelation (corr2)) match {
24       case (true, true) => KeyDist
25       case (false, false) => Inherent
26       case _ => VariableCost
27     }
28   }
29
30   override def adapt(env:  Environment, stats:  AdaptivePointStats):  Action = {
31     val stages = stats.stages
32     val rdd = stats.rdd
33     logInfo (s"${rdd}:  optimizing for taskImbalance")
34     val repr = getRepr (stages)
35     sourceOfImbalance (repr) match {
36       case Inherent =>
37         WarnAction (rdd.name, Inherent.toString)
38       case KeyDist =>
39         UPAction (rdd.name, "rangePartitioner")
40       case VariableCost =>
41         WarnAction (rdd.name, VariableCost.toString)
42       case NoImbalance =>
43         NOAction (rdd.name)
44     }
45   }
46 }
```

Figura B.4: Política de particionamento para fontes de desbalanceamento entre tarefas

```
1  import br.ufmg.cs.systems.sparktuner._
2  import br.ufmg.cs.systems.sparktuner.model._
3
4  object LocalityPolicy extends ApplicationPolicy {
5
6    private val replFactor:  Int = 3
7    private val localityTarget:  Double = 0.9
8
9    private def ln(n:  Double):  Double = {
10      scala.math.log(n) / scala.math.log(scala.math.E)
11   }
12
13   private def stageDelay(
14         numExecutors:  Int,
15         coresPerExecutor:  Int,
16         stage:  Stage
17      ):  Double = {
18      val taskRunTimes = stage.taskRunTimes
19      val avgTaskLength:  Double = (taskRunTimes.sum / taskRunTimes.size.toDouble)
20      val minSchedDelay = - (numExecutors / replFactor) *
21                           ln( (1 - localityTarget) / (1 + (1 - localityTarget)) )
22      (minSchedDelay / coresPerExecutor) * avgTaskLength
23   }
24
25   def beforeExec(env:  Environment):  Action = {
26      val numExecutors:  Int = env.executors.size
27      val coresPerExecutor:  Int = (env.executors.map (_.totalCores).sum /
28                           numExecutors).toInt
29     val schedDelay:  Double = env.stages.map (
30         stage => stageDelay(numExecutors, coresPerExecutor, stage)).max
31      UConfAction("spark.locality.wait", s"${schedDelay/1000}s")
32   }
33 }
```

Figura B.5: Política de aplicação para o *Delay Scheduling*

Glossário

API *Application Program Interface*. Descreve um conjunto de funcionalidades exportadas para o usuário de um sistema, através de uma interface de programação.

Cluster Um agregado de máquinas que servem de recurso para um ambiente de processamento paralelo e distribuído.

Correlação de Pearson Avalia o relacionamento linear entre duas variáveis, ou seja, a proporção das mudanças é levada em conta. Os valores possíveis variam entre -1 e 1, sendo que correlações fracas são próximas de zero.

Correlação de Spearman Avalia o relacionamento monotônico entre duas variáveis, ou seja, a proporção das mudanças não é levada em conta. Os valores possíveis variam entre -1 e 1, sendo que correlações fracas são próximas de zero.

DAG *Directed Acyclic Graph* (Grafo acíclico direcionado). Em computação paralela, representa o fluxo de dados e dependências de um programa.

Estágio Conjunto de RDDs e operações compostos por dependências *estreitas*, ou seja, que podem ser otimizados com *pipelining* (encadeamento) de transformações.

Fator No contexto de projeto fatorial, são parâmetros de um sistema sendo estudado, ou seja, o que varia. Por exemplo, o número de partições de uma execução é um candidato a fator no contexto de aplicações Spark.

Framework Ambiente (paralelo) de programação que executa sobre o recurso do *cluster*. Alguns sinônimos no contexto deste trabalho: **ambiente** e **plataforma**. Exemplos: Hadoop/M-R, Spark, Storm, Dryad, Flink, etc.

GC *Garbage Collection*. O gerenciamento de memória da maioria das linguagens que executam sobre uma máquina virtual (como *Java Virtual Machine*) é feito de

forma automática: um coletor de lixo é responsável por desalocar memória não mais referenciada pela execução.

Itemset No contexto do problema de mineração de padrões frequentes, significa um subconjunto de itens de uma transação. Os itemsets extraídos de um conjunto de transações são frequentes se sua ocorrência em toda a base ultrapassar um limiar predefinido, denominado suporte.

N-itemset Um itemset de tamanho n.

Pipelining Se refere à estratégia que, no contexto de DAGs de execução, faz a composição de operadores que não requerem comunicação em rede em uma única cadeia de funções. Essa estratégia é eficiente em evitar que dados intermediários sejam materializados desnecessariamente entre os operadores.

Política de aplicação Uma política de reconfiguração que toma decisões que impactam a aplicação como um todo, geralmente modificando uma configuração global.

Política de particionamento Uma política que atua nos pontos de adaptação da aplicação, afetando o particionamento da computação.

Ponto de adaptação Representa um ponto no DAG de aplicação no qual o particionamento do RDD pode ser naturalmente modificado. Na maioria dos casos, aparece nos RDDs imediatamente resultantes de uma etapa de *shuffle*.

Porcentagem de variação No contexto de um projeto fatorial, sempre está relacionado a uma métrica de avaliação e indica qual é a influência de um fator (ou interação entre um conjunto deles) no estudo da métrica. Uma porcentagem de variação elevada significa que o fator irá ter grande influência na métrica estudada. Naturalmente a influência pode ser positiva ou negativa.

Projeto fatorial Uma técnica que estuda o impacto dos fatores de um sistema através de uma métrica de avaliação. O projeto fatorial consegue estimar os efeitos atribuídos aos fatores e suas interações além de obter um modelo para estimativa da métrica em questão.

RDD *Resilient Distributed Dataset*. Abstração utilizada por Spark para representar coleções de dados distribuídas. Os vértices de um DAG representam os RDDs.

Shuffle Processo realizado por um *framework* de processamento paralelo onde os dados são reorganizados no *cluster* de acordo com um certo padrão de comunicação.

Speedup Métrica derivada que representa quantas vezes uma execução é mais rápida em comparação com outra.

Tarefa Unidade de escalonamento no *cluster*, seu propósito é aplicar uma função em uma partição do dado de entrada.

UDF *User Defined Function*. São funções definidas pelo programador que são aplicadas juntamente aos operadores durante a transformação de uma coleção de dados.