CEFET/RJ
DIRETORIA DE PESQUISA E PÓS-GRADUAÇÃO

UM *FRAMEWORK* ALGÉBRICO PARA *WORKFLOWS* DE ANÁLISE DE DADOS EM *APACHE SPARK*

João Antonio Ferreira

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Ciência da Computação, Centro Federal de Educação Tecnológica Celso Suckow da Fonseca CEFET/RJ, como parte dos requisitos necessários à obtenção do título de mestre.

Orientadores:
Eduardo Soares Ogasawara
Rafaelli de Carvalho Coutinho

Rio de Janeiro,
fevereiro de 2019

UM *FRAMEWORK* ALGÉBRICO PARA *WORKFLOWS* DE ANÁLISE DE DADOS EM *APACHE SPARK*

Dissertação de Mestrado em Ciência da Computação, Centro Federal de Educação Tecnológica Celso Suckow da Fonseca, CEFET/RJ.

João Antonio Ferreira

Banca Examinadora:

Presidente, Prof. Eduardo Soares Ogasawara, D.Sc. (orientador)

Prof. Rafaelli de Carvalho Coutinho, D.Sc. (co-orientadora)

Prof. Jorge de Abreu Soares, D.Sc. (CEFET-RJ)

Prof. Fabio Andre Machado Porto, D.Sc. (LNCC)

Prof. Leonardo Gresta Paulino Murta, D.Sc. (UFF)

Rio de Janeiro,
fevereiro de 2019

CEFET/RJ – Sistema de Bibliotecas / Biblioteca Central

F383 Ferreira, João Antonio
Um *framework* algébrico para *workflows* de análise de dados em *Apache Spark* / João Antonio Ferreira.—2019.
122f. + apêndices : il. color. , grafs. , tabs. ; enc.

Dissertação (Mestrado) Centro Federal de Educação Tecnológica Celso Suckow da Fonseca , 2019.
Bibliografia : f. 117-122
Orientadores : Eduardo Soares Ogasawara
Rafaelli de Carvalho Coutinho

1. Workflow. 2. Processamento de dados. 3. Framework. 4. Apache (Programa de computador). I. Ogasawara, Eduardo Soares (Orient.). II. Coutinho, Rafaelli de Carvalho (Orient.). III. Título.

CDD 658.53

Elaborada pela bibliotecária Mariana Oliveira CRB-7/5929-

DEDICATÓRIA

Dedico as pessoas que lutam para preservar a nossa frágil democracia.

AGRADECIMENTOS

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001. Agradece-se ao CNPq pelo financiamento parcial desta pesquisa.

Ao escrever estes agradecimentos relembro todo o caminho percorrido, do esforço empreendido e do apoio recebido. Seria impossível listar todas as pessoas que, de uma forma ou de outra, me ajudaram nesta caminhada, entretanto eu não poderia deixar de mencionar algumas pessoas.

Agradeço a minha família, em especial meus pais, que sempre lutaram para me proporcionar uma educação de qualidade.

Agradeço também as contribuições do meu orientador, Dr. Eduardo Soares Ogasawara que deu início as pesquisas sobre o tema abordado. Agradeço também a Dra. Rafaelli de Carvalho Coutinho, minha coorientadora, pelas inúmeras revisões no texto e pelas preciosas diretrizes na redação.

Agradeço a Dra. Maria Renilda Nery Barreto por todo o carinho e por ser a principal incentivadora para que eu enfrentasse o desafio de produzir uma dissertação de mestrado em Ciência da Computação.

Agradeço a equipe SOMA do CEPEL - Centro de Pesquisas de Energia Elétrica, em particular ao Dr. Renato Rocha, Dr. André Tomaz de Carvalho e ao Eng. Diogo R. C. Marques pelo apoio.

RESUMO

Um *framework* Algébrico para *Workflows* de Análise de Dados em *Apache Spark*

João Antonio Ferreira

A atividade típica de um cientista de dados envolve a implementação de diversos processos que caracterizam experimentos de análise de dados, modelados como *workflows*. Nestas análises há a necessidade de executar diversos códigos em diferentes linguagens de programação (Python, R, C, Java, Kotlin e Scala) em diferentes ambientes de processamento paralelo e distribuído. Dependendo da complexidade do processo e das inúmeras possibilidades para execução distribuída destas soluções, pode ser necessário gastar muita energia em diferentes implementações que afastam o cientista de dados do seu objetivo final, que é o de produzir conhecimento a partir dos grandes volumes de dados. Dentro deste contexto, este trabalho visa apoiar na solução de tal dificuldade ao propor a construção do *framework WfF*, concebido a partir de uma abordagem algébrica que isola a modelagem do processo da dificuldade de executar, de modo otimizado, tais *workflows*. Também foi criada uma linguagem agnóstica na forma de uma eDSL (*Embedded domain-specific language*) inspirada nos conceitos da MDA (*Model Driven Architecture*) para execução de *workflow* centrado nos dados (*dataflow*) e um gerador de código Scala para execução no *Apache Spark*. O uso de UDF (*User Defined Functions*) escritas em linguagens diversas, regidas por operadores algébricos (funções de segunda ordem da programação funcional) permite processamento otimizado de dados estruturados, semiestruturados e não estruturados ampliando o domínio de aplicações para além do *workflow* científico, podendo ser usado em *workflows* comerciais de análise de dados. As funcionalidades do ecossistema *Apache Spark* foram avaliadas no processo de otimização da execução de filtros (operador *filter*) que operam sobre UDF usando a API (*Application Program Interface*) *Catalyst* do *SparkSQL*, e os experimentos apontam a viabilidade desta abordagem.

Palavras-chave: *Workflow*; Spark; eDSL; MDA; *Framework*

ABSTRACT

An Algebraic Framework for Data Analysis Workflows in Apache Spark

The typical activity of a data scientist involves the implementation of various processes that characterize data analysis experiments. In these analyzes there is a need to execute several codes in different programming languages (Python, R, C, Java, Kotlin and Scala) in different parallel and distributed processing environments. Depending on the complexity of the process and the numerous possibilities for distributed execution of these solutions, it may be necessary to spend a lot of energy on different implementations that take the data scientist away from his ultimate goal of producing knowledge from large volumes of data. In this context, this paper aims to support this difficulty by proposing the construction of *WfF framework* conceived from an algebraic approach that isolates the process modeling from the difficulty of optimally executing such *workflows*. An agnostic language was also created in the form of eDSL - (Embedded domain-specific language) inspired by the MDA (Model Driven Architecture) concepts, for *dataflow* (data-centric workflow) execution and a Scala code generator for deploy in the Apache Spark environment. The use of UDF (User Defined Functions), written in diverse languages, governed by algebraic operators (second order functions of functional programming) allows optimized processing of structured, semistructured and unstructured data, extending the domain of applications beyond scientific, and can be used in commercial data analysis workflows. Spark ecosystem functionalities were evaluated in the process of filter optimization (filter operator) that operate on UDF using the SparkSQL Catalyst API, and the experiments demonstrated the feasibility of this approach.

Keywords: *Workflow*; Spark; eDSL; MDA; *Framework*

Sumário

Lista de Figuras

Lista de Tabelas

Listagens de programa

Lista de Abreviaturas

API *Application Program Interface*

APRF atributos peculiares à representação física

BIRA *Binary Input Relation Activity*

BPMN *Business Process Modeling Notation*

BPM *Business Process Management*

CIM *computation independent model*

CLI *Command Line Interface*

CNAE Classificação Nacional de Atividades Econômicas

core núcleo de processador

cores núcleos de processador

CRAN *Comprehensive R Archive Network*

css *cascading style sheet*

CSV *Comma Separated Values*

DAG *Directed Acyclic Graph*

DISC *Data Intensive Scalable Computing*

DNS *domain name system*

DSL *domain-specific language*

EBNF *extended Backus-Naur form*

eDSL *embedded domain-specific languages*

ETL *Extract, Transform and Load*

GIST *serviço do GitHub para hospedagem de trechos de código*

HDFS *Hadoop Distributed File System*

hosts computadores

host computador

html *hyper-text markup language*

HTTP *hyper-text transfer protocol*

IDE *Integrated Development Environment*

ISO *International Organization for Standardization*

JDBC *Java Database Connectivity*

JVM *Java Virtual Machime*

k8s *Kubernetes*

MDA *Model Driven Architecture*

MPI *Message Passing Interface*

NFS *Network File System*

nodes nós

node nó

OMG *Object Management Group*

OPM *Open Provenance Model*

PIM *platform independent model*

PPCIC Programa de Pós-graduação em Ciência da Computação

PSM *platform specific model*

REST *Representational State Transfer*

RPC *Remote Procedure Call*

RTE *Run-time Environment*

SaaS *Software as a Service*

SCM *Software Configuration Management*

SGBD Sistemas de Gerenciamento de Banco de Dados

SGW Sistemas de Gerenciamento de *Workflow*

SIMD *Single Instruction Multiple Data*

SOA *Service Oriented Architecture*

SQL *Structured Query Language*

TI Tecnologia da Informação

UDAF *Untyped User-Defined Aggregate Functions*

UDF funções definidas por usuário

UIRA *Unary Input Relation Activity*

UML *Unified Modeling Language*

W3C *World Wide Web Consortium*

WfF *Workflow Framework*

WfMC *Workflow Management Coalition*

WMS *Workflow Management System*

workers nós de trabalho

worker nó de trabalho

XML *eXtensible Markup Language*

YARN *Yet Another Resource Negotiator*

ZIP *Compressed file type*

Introdução

Contextualização

A resolução de diversos problemas científicos e comerciais de análise de dados requer o uso intensivo de recursos de processamento e armazenamento de dados. Visando uma execução eficiente, vários pesquisadores concentraram suas energias no desenvolvimento de otimizações voltadas, tanto às plataformas de hardware específicas (ambientes paralelos e distribuídos, GPU, TPU e FPGA) [Morcel et al., 2016], quanto aos algoritmos e métodos matemáticos [Nothaft et al., 2015]. Mais recentemente, com o crescimento da Internet, a popularização de smartphones, tablets e dispositivos de IoT (*Internet of Things*), houve um grande aumento da quantidade de fontes de dados e de volumes de dados gerados a cada minuto [1] [Gantz and Reinsel, 2011]. Para processar estes dados no tempo desejado, surgiram soluções escaláveis horizontalmente permitindo distribuir o processamento de grandes volumes de dados entre diversos computadores interligados em rede, como *clusters* e nuvem de computadores (*cloud*). Esta classe de soluções é chamada sistemas DISC (*data-intensive scalable compututing*) [Bryant, 2011]. Um dos modelos mais difundidos para esta distribuição de processamento é o paradigma *MapReduce* [Dean and Ghemawat, 2008], implementado em arcabouços (*frameworks*) tais como *Apache Hadoop* e *Apache Spark* [Zaharia et al., 2016]. Estes *frameworks* servem para execução de análises de dados em larga escala usando recursos de *clusters* ou de nuvens de computadores.

O principal benefício do modelo *MapReduce* é a escalabilidade horizontal, onde os dados, por maior que seja seu volume, poderão ser particionados e distribuídos entre os computadores, proporcionando a vantagem da localidade do dado [2], de forma que

[1] Por exemplo, apenas um dos sistemas de monitoramento da usina de Itaipu gera 23.76 milhões de registros de série temporal por minuto. Tais registros contém dados de *timestamp, tag* e valor para medidas obtidas de acelerômetros, proximetros, termopares, etc.

[2] Localidade do dado refere-se ao conceito de que dados mais próximos da CPU em relação ao tempo necessário para seu acesso exigem menos custo de processamento. Em ordem crescente de custo (latência) tem-se, de forma aproximada: cache L1, cache L2, cache L3, SSD, HD, Rede.

o programa possa processar, em cada nó (node) da rede, apenas um pequeno conjunto de dados disponível em sua vizinhança. Após a finalização deste processamento, informações de agregação podem ser trocadas entre os nodes para consolidação dos resultados.

Motivação

Neste contexto de análise de dados, têm-se disponíveis diversas bibliotecas escritas em Python (NumPy, SciPy, Scikit-learn) e em pacotes R no *Comprehensive R Archive Network* (CRAN) capazes de prover aos cientistas uma classe de soluções numéricas para análise exploratória, cálculo vetorial, álgebra linear, métodos estatísticos, visualização de dados e aprendizado de máquina. O uso de tal ferramental nos *frameworks* como Apache Spark comumente recai na implementação de funções definidas por usuário (UDF), de modo a viabilizar a análise de dados em larga escala, a qual faz uso das soluções escritas em Python e R.

No entanto, apesar de sua importância, o uso de UDF traz desafios no processo de otimização de execução, visto a dificuldade de se estabelecer semântica clara sobre seus comportamentos que, via de regra, possuem códigos *ad-hoc* relacionados ao domínio. Com a falta de conhecimento sobre a semântica da operação, *frameworks* como o *Spark* são incapazes de otimizar sua execução [Armbrust et al., 2015]. Rheinländer et al. [2017] dedicaram um estudo extenso a questão da otimização de execução de UDF em *dataflow* (*workflow* centrado em dados), levantando os principais desafios relacionados e apontando abordagens possíveis para solução, dentre as quais, destaca-se a álgebra de *workflow* [Ogasawara et al., 2011].

Uma forma de anotar semântica em UDF dentro de *dataflow* é usar os conceitos de álgebra de *workflow* [Ogasawara et al., 2011] com dados de proveniência retrospectiva [3]. O *Spark* permite que se obtenha métricas e informações sobre o ambiente de execução via a *Application Program Interface* (API) *SparkListener* [4], e neste caso é possível filtrar e processar informações via *SparkListener* e armazenar tais informações como proveniência retrospectiva em um repositório de apoio. Com isso, podem-se ampliar as

[3] Proveniência retrospectiva é o mecanismo de armazenar de forma segura informações que possam determinar se os resultados de um experimento é confiável, reprodutível, e como dar crédito a seus criadores ao reutilizá-lo

[4] *SparkListener* é um mecanismo para interceptar eventos do escalonador do *Spark*. Estes eventos são emitidos durante a execução de um aplicação *Spark*. O projeto *sparkMeasure*, disponível no GitHub, implementa um coletor de métricas de desempenho do *Apache Spark* usando *SparkListener* e foi utilizado na avaliação experimental - Capítulo 4

capacidades de otimização de UDF considerando o custo de processamento em termos de tempo médio de execução para cada tupla da entrada. Informações da quantidade de *threads* e de memória de acesso randômico também podem ser coletadas para apoiar otimização relacionada à materialização de resultados intermediários.

Além disso, o processo de desenvolvimento de *workflows* centrados nos dados (*dataflow*) envolve conhecimento sobre o domínio do problema e sobre os métodos e algoritmos mais adequados à solução. Trata-se de um processo de desenvolvimento de software, e como tal, possui seu próprio ciclo de vida, com diversas atividades a serem desempenhadas por cientistas e engenheiros de dados. Estas atividades podem ser classificadas em : *i)* coleta de requisitos; *ii)* atividades de análises; *iii)* atividades de projeto e desenvolvimento da solução; *iv)* atividades de testes e validação; *v)* atividades de implantação. Cada uma dessas atividades impõe seus próprios desafios, e pode ser mais ou menos relevante dependendo do problema a ser solucionado. Por exemplo, as atividades de implantação tem maior importância em *dataflows* comerciais, pois entram no processo produtivo normal da empresa após os testes.

Para diminuir os custos de produção destas soluções, além de melhorar a qualidade e a longevidade do software, podem-se usar os conceitos de *Model Driven Architecture* (MDA) que define como objetivo principal usar linguagem de modelagem como linguagem de programação [Frankel, 2003]. Nesta abordagem é possível esconder detalhes de implementação e definir os requisitos do *dataflow* em um alto nível de abstração, tanto para os aspectos estáticos quanto os dinâmicos da aplicação. Na MDA, esta abstração é denominada PIM (*Platform Independent Model*) embora a MDA também aceite artefatos do PSM (*Platform Specific Model*) [Singh and Sood, 2009]. Juntos, o PIM e o PSM contêm todos os detalhes necessários para a execução de uma dada aplicação. A vantagem desta abordagem é a independência de plataforma e de linguagem de programação que possui valor adicional na plataforma *Apache Spark*, pois aceita diversas linguagens de programação diferentes: *i)* fracamente tipadas (Python e R) e *ii)* fortemente tipadas, internas a *Java Virtual Machime* (JVM) (Kotlin, Java e Scala). Em resumo, com a MDA, pode-se especificar a aplicação usando uma linguagem de modelagem de alto nível e usar as técnicas de reusabilidade na geração de código para produzir programa executável para o ecossistema *Apache Spark* em quaisquer das linguagens aceitas.

Objetivo

Dado que temos este problema de alto custo computacional para execução de *dataflow* em ambiente *bigdata*, este trabalho propõe um método para otimização das execuções de *workflows* de análise de dados que possam ser escritas como UDF em ambientes de processamento distribuído em larga escala (sobre o *Apache Spark*). Além disso, estas UDF poderão usar bibliotecas e códigos escritos em linguagens de análise de dados (Python, R, Java, Kotlin e Scala). Para viabilizar a utilização de tais otimizações foi desenvolvido um *framework*. É importante observar que dados de proveniência retrospectiva de execuções anteriores do mesmo *dataflow* serão usados para anotar as semânticas nas UDF.

O *framework* proposto será referenciado neste documento como *Workflow Framework* (WfF).

Aplicações e benefícios do *framework*

É possível classificar em dois grandes grupos os domínios de aplicação aos quais o *framework* para *workflow* de análise de dados em larga escala pretende apoiar. São eles : *i)* científicos; *ii)* comerciais. Problemas científicos diferem em alguns aspectos de problemas comerciais, o que tem levado a comunidade científica a desenvolver soluções seguindo esta dicotomia. Ocorre, porém, que existem muitos pontos em comum que podem ser atendidos pelo *framework* proposto.

Algumas das categorias de *workflow* que podem tirar vantagem do *framework* proposto, incluem: *i) Extract, Transform and Load* (ETL) que tanto podem ser científicos quanto comerciais; *ii)* varredura de parâmetros (Otimização de Hiper-parâmetros) para aprendizado de máquina; *iii)* análises complexas envolvendo dados estruturados, semi-estruturados e não estruturados distribuídos na WEB e nas Intranets; *iv) workflows* de integração de aplicações legadas de análise de dados escritos em R, Python, Java, GO, C, C++, que podem ser definidos em um único *dataflow* e otimizados algebricamente [Ferreira et al., 2017]; *v)* aplicações de análise de dados da indústria 4.0 [5], minerando conhecimento a partir de dados provenientes de sensores (*IoT*) para apoiar a gestão da manutenção de ativos (máquinas e equipamentos industriais), identificando e diagnosticando falhas; *vi) workflow* para apoiar *Computation Offloading*, que é uma técnica

[5] Industria 4.0 refere-se a quarta revolução industrial onde agregam-se aos processos industriais a inteligência artificial, os sistemas ciber-físicos, a computação em nuvem e a Internet das Coisas (IoT)

de transferência da execução de tarefas computacionais com uso intensivo de recursos de processamento para uma plataforma externa, como um *cluster*, uma grade (*grid*) ou uma nuvem (*cloud*), para economizar recursos em dispositivos com baixa capacidade, tais como dispositivos remotos de aquisição de dados (*IoT*) ou telefones celulares inteligentes (estes últimos devido ao consumo excessivo de carga de bateria [Deng et al., 2015]).

Como subproduto deste trabalho foi desenvolvido uma *domain-specific language* (DSL) em texto-plano (*plain-text*) para facilitar a utilização do *framework* por parte dos usuários. Assim é possível gerar automaticamente o código em linguagem Scala, correspondente a um dado *workflow*, para execução no *Spark*.

Esta linguagem de especificação de *workflow* é definida por uma DSL (*Domain-specific Language*) interna (eDSL) [6] escrita em linguagem Kotlin [Jemerov and Isakova, 2017]. Como requisito, a gramática da linguagem deve refletir as entidades e os relacionamentos encontrados no *dataflow* (relações, esquemas, operadores, definição de Interfaces para UDF externas, via assinatura dos métodos, e definição de materialização de resultados). Além disso, a gramática da linguagem deve permitir a separação das responsabilidades entre os cientistas de dados e os engenheiros de dados[7], ou seja, a gramática permite que se crie artefatos descrevendo o *workflow* de uma forma abstrata, além de ligações (*binding*) entre a definição abstrata e as materializações/implantações/execuções em um ambiente de *runtime* específico, de acordo com os requisitos da MDA. A abordagem de convenção sobre configuração (*Convention over configuration*) [8] é usada no *framework* WfF para simplificar a definição dos detalhes de infraestrutura relacionados a uma dada execução do *workflow*, por parte do usuário.

Outro benefício do *framework* (WfF), que está alinhado ao conceito *DevOps* [9], é a separação de conceitos (*Separation of concerns*) [Dijkstra, 1982], provida pela gramática proposta na *embedded domain-specific languages* (eDSL) do *framework* WfF, que usa as ideias da MDA permitindo que cientistas de dados direcionem o foco para os modelos, deixando detalhes de plataforma de execução com engenheiros de dados e outros

[6] eDSL - *Embedded domain-specific languages* [Felleisen et al., 2018] e [Alexandrov et al., 2016]

[7] Em ambientes de Tecnologia da Informação (TI), os administradores de banco de dados (DBAs) tratam da infraestrutura e os administradores de dados (DAs) juntamente com os Desenvolvedores tratam dos modelos. A situação é análoga com Engenheiros de dados e Cientistas de dados

[8] Este conceito foi cunhado por Dumbill [2005] criador do Ruby on Rails para simplificar o desenvolvimento de software aumentando a produtividade.

[9] DevOps refere-se a um conjunto de práticas para integração entre as equipes de desenvolvimento de software, operações, infraestrutura e de controle de qualidade e a adoção de processos automatizados para produção de aplicações e serviços de TI

profissionais de TI.

Resultados

Os experimentos realizados com um *workflow* baseado no *benchmark TPC-H* confirmam a viabilidade da abordagem de otimização proposta neste trabalho, já que os resultados apresentaram um ganho em torno de 15% em termos de tempo total de execução do *dataflow* baseado na *query* `q13` [10].

Também foi realizada uma prova de conceito para validar a transparência em relação às características do sistema operacional e das materializações dos *datasets* referenciados no *workflow*, ou seja, valida a transparência provida pela MDA em relação a especificação das fonte de dado (*datasource*) de origem e destino.

Limitações do *framework*

Quando consideramos a implementação do *framework*, o presente trabalho envolve várias áreas de conhecimento e portanto foi necessário definir algumas limitações no escopo, que são listadas abaixo:

O tema otimização de *dataflow* é extenso e assim avaliou-se as funcionalidades do ecossistema *SparkSQL*/*Hadoop* no processo de otimização da execução de filtros (operador *filter*) que operam sobre UDF usando a API *Catalyst* do *SparkSQL* [11]. Além disso, não foram investigados otimizações de ponta a ponta em *dataflow* contendo filtros e junções como analizado por Hellerstein [1992].

Na avaliação experimental apresentada no Capítulo 4, os dados de proveniência foram instrumentados, ou seja definidos *hard-coded* , mas é possível coletá-los usando a API *SparkListener*, como mencionado anteriormente.

Organização do trabalho

Este trabalho está organizado em seis capítulos. No Capítulo 1 - *Referenciais Teóricos*, são apresentados conceitos gerais necessários para o entendimento do problema

[10] Sobre a escolha da *query* `q13`, veja as Seções 4.6 - Experimentos para avaliação da otimização algébrica e
4.8 - Ameaça a validade.

[11] A API *Catalyst* é um *framework* de software que apoia o desenvolvimento de otimização de expressões de qualquer natureza, incluindo relações e seus esquemas, participantes de um *dataflow*. Exemplo de classe usando o *Catalyst* pode ser visto na Seção 3.6

e da metodologia aplicada, além dos detalhes da API de otimização do *SparkSQL* e da cadeia de ferramentas para geração de código na linguagem Scala. No Capítulo 2 - *Trabalhos Relacionados*, os trabalhos publicados relacionados ao tema da pesquisa são explorados, para que produza contribuição relevante considerando o estado da arte. Em seguida, no Capítulo 3 - framework WfF - *Desenvolvimento da Solução*, apresenta-se a abordagem adotada e o desenvolvimento da solução incluindo a descrição da implementação. O Capítulo 4 - *Avaliação Experimental* traz a avaliação com os resultados experimentais em ambiente de computação distribuída (*cluster*) que corroboram a hipótese de viabilidade da otimização via Catalyst de *dataflow* com UDF regidas por operadores *Filter* dispostos de forma contígua. No Capítulo 4.9 apresentam-se as conclusões e as questões em aberto que podem ser objeto de trabalhos futuros para expandir o *framework* em termos de funcionalidades e em termos de melhoria na robustez e meta-gestão.

(Página intencionalmente deixada em branco)

Capítulo 1 Referenciais Teóricos

Chronis et al. [2016] abordam a importância da elasticidade no processamento de *bigdata* e o quanto a computação em nuvem tem contribuído para o que se convenciona chamar computação elástica (*Elastic Computing*) ou computação adaptativa, na qual recursos são alocados dinamicamente de acordo com a demanda. Os autores citam que soluções para problemas intensivos em dados devem considerar o compromisso entre tempo de execução e custo de infraestrutura computacional. A Figura 1, retirada de [Chronis et al., 2016], ilustra este efeito.

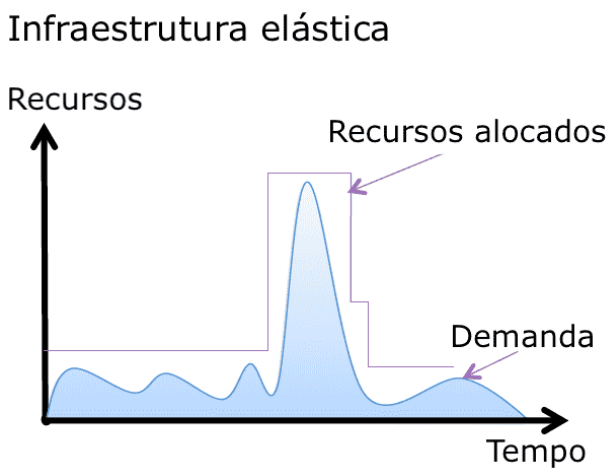


Figura 1: Elasticidade na alocação de recursos computacionais. A imagens ilustram as características de alocação de recursos computacionais durante o ciclo de vida de uma aplicação intensiva em dados. A demanda por recursos pode variar drasticamente forçando alocação de muitos recursos em um curto espaço de tempo.

Vários provedores de infraestrutura fornecem serviços de computação em nuvem com estas características. Além disso, soluções de código aberto como OpenNebula One, OpenStack, CloudStack e Eucaliptus [1] apoiam a criação de infraestrutura elástica em nuvens computacionais privadas. Os *workflows* intensivos em dados baseados em *Spark* são amplamente implementados nessa plataforma de processamento de *bigdata*

[1]Os links correspondentes no GitHub são: OpenNebula - `https://github.com/OpenNebula/one`, OpenStack - `https://github.com/openstack`, CloudStack - `https://github.com/apache/cloudstack` e Eucaliptus - `https://github.com/eucalyptus/eucalyptus`

[Zheng et al., 2017].

Com o problema de infraestrutura para computação de alto desempenho equacionado resta ainda o problema do software capaz de atender aos desafios relacionados à análise de dados que se apresentam. Este é o foco principal deste trabalho, que pretende realizar as otimizações de *workflow* de análise de dados sobre o *Apache Spark*. Neste contexto, o presente capítulo apresenta conceitos gerais de *Spark*, sistemas de gerência de *workflow*, UDF, álgebra relacional, álgebra de *workflow* [Ogasawara et al., 2011], proveniência retrospectiva, Linguagem Específica de Domínio, MDA e importantes conceitos para entendimento do trabalho.

1.1 *Apache Spark*, *SparkSQL* e *Catalyst*

O *Spark* é um *framework* que possibilita a execução das tarefas paralelizáveis de forma distribuída em máquinas *multi-core* ou *clusters* YARN/Mesos, com ênfase no processamento em *pipeline* de atividades, com alocação de arquivos intermediários primordialmente em memória. O *Spark* pode ser classificado como um sistema *Data Intensive Scalable Computing* (DISC), pois atende aos quatro princípios chave para este tipo de sistema [Bryant, 2011]. Os códigos podem ser escritos em uma das linguagens para as quais o *framework* oferece sua API, incluindo: Scala, Java, Python e R. O *framework* possui arquitetura modular que permite a inclusão de componentes adicionais ao núcleo do *framework* (*Spark Core*), tais como: *SparkSQL, GraphX, Spark MLlib* e *Spark Streaming*. O *Spark* se integra com *Hive, HBase, HDFS, Parquet* e todo o ecossistema *Hadoop*. Finalmente, o armazenamento de arquivos intermediários em memória faz com que a execução ocorra dezenas ou até centenas de vezes mais rápida que uma implementação equivalente em *Apache Hadoop* [Zaharia et al., 2016].

Dos cinco componentes principais quatro deles são relevantes para este trabalho: *Spark Core*, *GraphX*, *SparkSQL* e a *Spark MLlib*. O *Spark Core* implementa a base do mecanismo de execução de tarefas separando as *transformações* das *ações* sobre os RDD (*Resilient Distributed Datasets*) [Zaharia et al., 2012] e, mais recentemente, sobre os *dataframes*. As operações de *transformação* são de avaliação tardia e podem ser invocadas várias vezes antes que uma *ação* seja executada. Tal abordagem permite que o *Spark* escolha a melhor forma de executar as transformações considerando a quantidade de memória, os dados de localidade, o número de computadores no *cluster*, o número de *cores* nos processadores e as otimizações de código por meio do *Catalyst*

[Armbrust et al., 2015].

O *GraphX*, descrito em [Xin et al., 2013], é um componente do *Spark* que realiza computação paralela de grafos e estende o RDD do *Spark* introduzindo uma nova abstração: um multigrafo direcionado com propriedades anexadas a cada vértice e cada aresta. Além disso, o *GraphX* inclui uma coleção crescente de algoritmos e *builders* para simplificar tarefas de análise de grafos que permite executar caminhamento em grafos em larga escala e que é utilizado, por exemplo, em problemas de análise de trajetória em aplicações de logística, transporte e cidades inteligentes [Almeida et al., 2016].

Já o *SparkSQL* faz também uso do *Catalyst* para otimizar geração de código em tempo de execução a partir de expressões e comandos *Structured Query Language* (SQL). O *SparkSQL* permite acesso a *datasets* usando uma interface baseada em álgebra relacional [Armbrust et al., 2015]. Trata-se de um módulo para processamento de dados estruturados. As interfaces do *SparkSQL* fornecem informações adicionais sobre a estrutura dos dados e a computação que está sendo executada. Com o *SparkSQL* pode-se fazer uso, no mesmo programa, da linguagem SQL declarativa e APIs de acesso a dados em *DataFrame*/*Dataset* de modo imperativo/procedural.

A *Spark MLlib* é a biblioteca de aprendizado de máquina (ML) do *Apache Spark*. Seu objetivo é tornar a aprendizagem de máquina escalável e fácil de usar. A *Spark MLlib* [Meng et al., 2016] fornece algoritmos de aprendizagem consolidados, tais como: classificação, regressão, agrupamento e filtragem colaborativa. Também fornece ferramentas para pré-processamento de mineração de dados, tais como seleção, redução de dimensionalidade, transformação de dado, e extração de características. Outra ferramenta da *Spark MLlib* é a de construção, avaliação e ajuste de *pipelines* de aprendizado de máquina. Algoritmos e APIs utilitárias para álgebra linear, estatísticas e tratamento de dados também são fornecidos pela *Spark MLlib*. Esta API é baseada na API de *DataFrame*/*Dataset* que se vale das otimizações do *Catalyst* disponível com *SparkSQL*, citado anteriormente.

O *Apache Spark*, que foi inicialmente concebido para executar processos distribuídos em larga escala com tolerância a falhas, mais recentemente [2] ganhou dois outros módulos: *SparkR* e *GraphFrames*. O *SparkR* [Venkataraman et al., 2016] permite executar um *dataflow* a partir de uma seção no programa R, tendo como único pré-requisito a ins-

[2]Outra funcionalidade recente, ainda em versão *Preview* para Spark 2.4+ e Scala 2.11 no Linux, é o *TensorFrames (TensorFlow on* Spark *Dataframes)* que permite manipular *DataFrames* a partir de programas *TensorFlow* escritos em Scala ou Python, integrando *Deep Learning* com o ecossistema *SparkSQL*/*Hadoop* `https://github.com/databricks/tensorframes`

talação do pacote *SparkR* no computador *host* (`install.packages(c("SparkR"))`. Isto abre possibilidades de iniciar uma análise de dados exploratória a partir da REPL (*Read-Eval-Print-Loop*) do R. Com esta funcionalidade os usuários de R podem fazer análises exploratórias usando recursos de *clusters* e nuvens ampliando muito os casos de uso possíveis.

Outro módulo, disponível como *Spark Module* é o *GraphFrames* descrito por Dave et al. [2016] que permite especificar um *dataflow* de análise de dados de relacionamentos entre objetos usando grafos via linguagem declarativa, similar a SQL, em problemas que envolvam nós e vértices. No *GraphFrames* tem-se disponível uma DSL com representação simbólica para vertices e arestas, facilitando o entendimento. Trata-se de uma evolução natural do *GraphX*, já citado anteriormente, e que oferece uma interface declarativa mais simples e que integra com a API *MLlib*. Um exemplo simples pode ser visto nas Figuras 2, 3 e 4 [3].

Com a DSL do *GraphFrames* é possível, por exemplo, especificar a consulta `Pessoas que talvez alguém conheça` pela cadeia de caracteres: $(a) - [e] - > (b); (b) - [e2] - > (c); !(a) - [] - > (c)$. Nesta representação (a) significa um certo vértice a e $[e]$ significa uma dada aresta e. Direções das arestas e operadores lógicos também podem ser especificados, enriquecendo a linguagem de consulta simbólica. Veja na na Figura 5 como podemos usar esta DSL e a API *GraphFrames* para listar as pessoas que Alice (cujo id é "a") possa conhecer, através de um contato indireto (cujo vértice é "(b)"). As pessoas pessoas que Alice possa conhecer são representadas pelo vértice "(c)". Observe que $!(a) - [] - > (c)$ exclui as ligações diretas entre Alice e estas pessoas, ou seja o operador $;$ adiciona um seleção conjuntiva a expressão à sua esquerda.

O *Spark* permite que um *dataflow* seja especificado usando programação funcional em uma estrutura conhecida como *fluent API*, onde operações (tarefa/atividade) são funções de segunda-ordem e podem ser encadeadas uma após a outra. Isto pode ser observado na Figura 3 com *GraphFrames* e na Figura 6 que representa um *workflow* envolvendo análise de dados em tabelas de um banco de dados relacional e em dados semiestruturados obtidos da WEB.

Uma funcionalidade chave do ambiente *Spark* para este trabalho é o fato deste permitir otimizações de código por meio da API *Catalyst*, que é um *framework* de software para apoiar a otimização de expressões de qualquer natureza, incluindo aquelas refe-

[3] Outros exemplos podem ser vistos no repositório `https://github.com/joao-parana/graphframes/tree/master/example`

```scala
/**
 * Graph of friends in a social network.
 */
def friends: GraphFrame = {
  // Vertex DataFrame
  val v = spark.createDataFrame(List(
    ("a", "Alice", 34),
    ("b", "Bob", 36),
    ("c", "Charlie", 30),
    ("d", "David", 29),
    ("e", "Esther", 32),
    ("f", "Fanny", 36),
    ("g", "Gabby", 60)
  )).toDF("id", "name", "age")
  // Edge DataFrame
  val e = spark.createDataFrame(List(
    ("a", "b", "friend"),
    ("b", "c", "follow"),
    ("c", "b", "follow"),
    ("f", "c", "follow"),
    ("e", "f", "follow"),
    ("e", "d", "friend"),
    ("d", "a", "friend"),
    ("a", "e", "friend")
  )).toDF("src", "dst", "relationship")

  // Create a GraphFrame
  GraphFrame(v, e)
}

val g = friends
```

Figura 2: *GraphFrames* - Exemplo de grafo criado representando as ligações entre pessoas numa rede social. Cria-se um DataFrame para vertices e outro para arestas e com isso é possível criar o *GraphFrame*.

```scala
/**
 * Returns a list containing the most connected persons.
 */
def getMostConnected(g: GraphFrame, topN: Int): Dataset[Row] = {
    val gInDegrees = g.inDegrees
    g.vertices.join(gInDegrees, "id").orderBy(desc("inDegree")).limit(topN)
}
```

Figura 3: *GraphFrames* - Método `getMostConnected` em linguagem Scala, para gerar a lista com as pessoas que possuem o maior número de conexões na rede.

rentes às relações e seus esquemas, participantes de um *dataflow*. As interfaces do *SparkSQL* fornecem informações adicionais sobre a estrutura dos dados em catálogo e a computação que está sendo executada. Internamente, o *SparkSQL* usa essas informações extras para realizar as otimizações. Os componentes *MLlib* e *GraphFrames*

```
// Lista as 5 pessoas mais seguidas na rede
val mostConnected = getMostConnected(g, 5)
mostConnected.show()
```

Figura 4: *GraphFrames* - Listando as 5 pessoas mais seguidas na rede usando o *GraphFrames*.

```
// Pessoas que talvez a Alice conheça
val peopleYouMayKnow = g.find("(a)-[e]->(b); (b)-[e2]->(c); !(a)-[]->(c)").
  filter("a.id = 'a'").select("c.id", "c.name")
peopleYouMayKnow.show()
```

Figura 5: *GraphFrames* - Listando as pessoas que Alice possa conhecer, através de um contato indireto, por outra pessoa.

também podem ser otimizados com *Catalyst* no contexto de *dataflows*, já que foram escritos sobre a API *DataFrame*/*Dataset* do *SparkSQL*.

Para facilitar o acesso às funcionalidades do *Catalyst* foram criados na versão 2.2 do *Spark*, vários pontos de extensão [4], permitindo que desenvolvedores façam suas próprias implementações. Isso possibilita a personalização do ambiente *Spark* com o uso de regras próprias para otimização nos seguintes escopos: *i)* analisador gramatical (*parser*); *ii)* analisador (*analyzer*); *iii)* otimizador (*optimizer*); *iv)* estratégia de planejamento físico (*physical planning strategy*). Na Seção 1.1.1, podem ser vistos os detalhes de todo o processo e das funcionalidades da API de otimização do *SparkSQL* e do *Catalyst*. Estes pontos de extensão são suficientes para desenvolver uma gama de otimizações personalizadas no *SparkSQL*.

Além da otimização de código, o *SparkSQL* permite outras funcionalidades graças à inferência de esquema, como por exemplo, *JOIN* entre tabelas de Sistemas de Gerenciamento de Banco de Dados (SGBD) relacionais e arquivos JSON ou XML. Além disso, ele apoia a criação de UDF e seu respectivo registro no catálogo permitindo ao componente *Catalyst* considerá-las no processo integrado de otimização do *dataflow*.

Segundo Armbrust et al. [2015] para usar o *Catalyst* é necessário lançar mão de características específicas da linguagem Scala, tais como, funções parciais, correspondência de padrões, *quasiquotes* e tipagem implícita. Assim, é possível criar uma DSL interna (eDSL) para traduzir o código Scala para SQL [Cheney et al., 2013] e converter um *dataflow* de código Scala imperativo para código declarativo, tal como SQL, e

[4] Os pontos de extensão foram implementados a partir do épico documentado em *Add hooks and extension points to Spark* - `https://issues.apache.org/jira/browse/SPARK-18127`.

```
employees.
      join(departments, "DeptId").
      select("DeptName", "LastName", "getTweets(DeptId) as Tweets").
      orderBy("DeptName").
      show()
```

Figura 6: *Dataflow* especificado como paradigma funcional em uma *fluent API*. No exemplo, uma UDF pesquisa dados semiestruturados na WEB e combina com dados estruturados do Banco de Dados.

otimizá-lo usando o *Catalyst*. Esta é abordagem que foi adotada neste trabalho.

1.1.1 Os detalhes da API de otimização do *SparkSQL* e do *Catalyst*

A otimização de consultas (*queries*) é um assunto estudado desde a década de 1970 em tecnologia de bancos de dados relacionais. Elmasri and Navathe [2015], no livro "*Fundamentals of Database Systems*", apresentam no capítulo 19 "*Algorithms for Query Processing and Optimization*"as técnicas usadas internamente pelos gerenciadores de banco de dados relacionais para analisar, processar, otimizar, e executar consultas de alto nível.

O processo típico de otimização de *query* passa pelas etapas apresentadas na Tabela 1 com os resultados intermediários correspondentes.

Etapa	Processo	Resultado
1	análise léxica (*scanning*), *parsing* e validação	AST
2	otimizador de *query*	plano de execução (*query tree*)
3	gerador de código para *query*	código binário
4	processador de *runtime* para *query*	resultado de consulta

Tabela 1: Processamento e otimização em consultas de Banco de Dados

Na etapa 1, o *scanner* valida os *tokens*, o *parser* faz a verificação sintática da *query* e o validador verifica todos os atributos e a respectiva semântica contra o esquema das relações no banco de dados. Na etapa 2, o otimizador procura gerar o melhor plano de execução possível que possa ser criado de uma forma eficiente usando recursos computacionais limitados a um certo valor viável (a otimização não pode consumir, por exemplo, mais recursos computacionais que a própria *query*). O otimizador gera uma *query tree* ou um *query graph* dependendo da implementação. Na etapa 3, o código é gerado a partir do plano de execução em um formato binário que possa ser executado pelo processador de *runtime* do DSL.

Na etapa de otimização é preciso determinar os modos alternativos de avaliação de

uma determinada consulta, encontrando as expressões equivalentes e considerando os diferentes algoritmos para cada operação e a diferença de custo entre as diversas maneiras equivalentes [5] de obter os resultados da consulta. A estimativa de custo das operações depende de informações estatísticas sobre as tabelas que o DSL precisa manter, como por exemplo o número de tuplas, o número de valores distintos para atributos de junção (*JOIN*), dentre outros. É necessário também estimar estatísticas para resultados intermediários[6], para calcular o custo de expressões complexas. Em geral, os DSL expõem comandos para que o DBA (*Database Administrator*) possa fazer ajustes no modelo físico para melhorar o desempenho de uma consulta que esteja usando muitos recursos computacionais. Estes comandos permitem avaliar o plano de execução para uma dada *query* ou forçar o DSL a recalcular as métricas estatísticas.

Elmasri and Navathe [2015] apresentam um total de 12 regras de transformação sobre operadores da álgebra relacional que não alteram a semântica da *query* e que podem ser usadas em conjunto para otimizar uma dada consulta. A regra transformada é equivalente à original, porém com custo de execução presumidamente menor. É importante observar que *queries* podem possuir *subqueries* e estas transformações permitem unificá-las alterando-se os predicados de forma adequada e usando operações de junção, projeção, etc. Como este processo é baseado na álgebra relacional, todas as propriedades matemáticas da teoria de conjuntos podem ser aplicadas de forma segura para melhorar o plano de execução e isto inclui, por exemplo, a comutatividade da seleção (operador *SELECT*), da interseção, da união e da junção natural. O inconveniente é a quantidade enorme de combinações possíveis de planos de execução equivalentes que podem ser geradas na etapa de otimização. Para resolver este problema, usam-se heurísticas para reduzir o espaço amostral. Todo este conhecimento sobre otimização em *queries* relacionais pode ser usado no *SparkSQL* e no *Catalyst*. De fato, o *Catalyst* possui mais de 60 regras já criadas que usam conhecimento de teoria de banco de dados e teoria de compiladores. Além disso, o *SparkSQL* disponibiliza, via *Catalyst*, um mecanismo para adicionar pontos de extensão (*extension points*) com otimizações personalizadas e esta funcionalidade será usada neste projeto para implementação do *framework* algébrico. Segue uma descrição do processo de otimização do *Catalyst*. Se-

[5] Resultado equivalente significa que a semântica da consulta (*query*) original foi mantida e que é garantida a equivalência do resultado obtido, variando apenas os custos de execução.

[6] Uma cláusula *WHERE*, por exemplo, pode reduzir a quantidade de tuplas processadas a 10% do total original alterando drasticamente o tempo de processamento no caso em que os dados são filtrados antes de uma junção.

rão usados os termos *SparkSQL* e *Catalyst* para se referir a infraestrutura de otimização do *Apache Spark*, pois um é subprojeto do outro.

No diagrama da Figura 7, ilustram-se os componentes do processo de otimização e geração de código do *SparkSQL*. O *Catalyst* processa relações oriundas de comandos SQL, mas também processa *LogicalPlan* oriundos de *LocalRelation*, sem dependência com as APIs RDD, SQL ou *DataFrame*. Isto é útil para instrumentação com testes unitários e desenvolvimentos *stand-alone*, sem toda a infraestrutura do *Spark*, a qual fica disponível via objeto *SparkSession*.

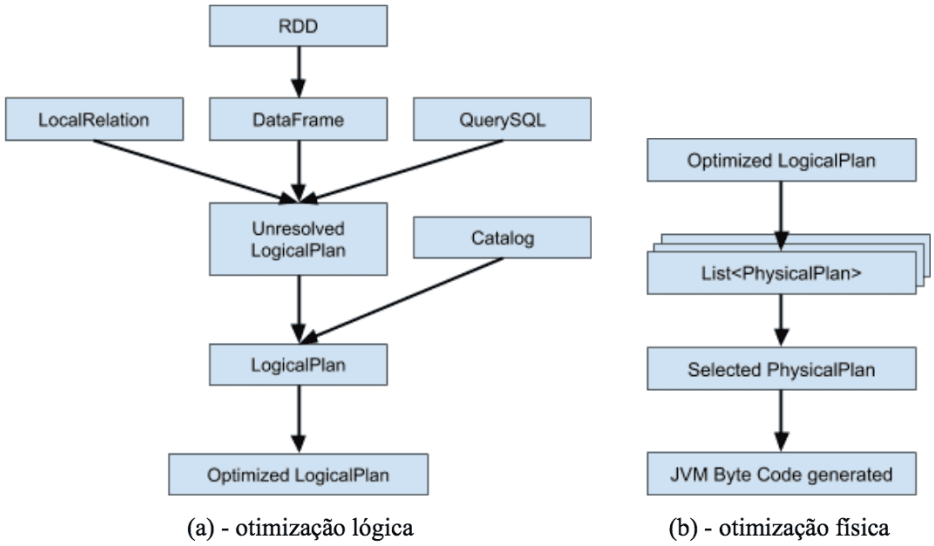


Figura 7: Artefatos *Catalyst*. (a) partindo de um RDD, *LocalRelation*, *Dataframe* ou uma *query* SQL, obtêm-se os diversos artefatos ao realizar um processo de otimização lógica. (b) partindo um plano lógico otimizado um processo de otimização física permite gerar código otimizado considerando informações de custo computacional.

O diagrama da Figura 7(a) mostra como se obtém as otimizações lógicas baseadas em regras (*rule-based optimization*) escritas com base na teoria de compiladores e teoria de bancos de dados. As otimizações físicas do diagrama da Figura 7(b) implementadas no *Spark* são baseadas no custo da computação (*cost-based optimization*) e considera as informações de localidade dos dados.

Em relação a geração de *byte-code* para a JVM, o *Spark* utilizava, nas versões mais antigas, o modelo de iterador Volcano (*Volcano iterator model*) [Graefe, 1994], que funciona como um iterador clássico muito simples e flexível [7], mas apresenta desempenho ruim em CPUs modernas devido à chamada de funções que impede otimizações dos

[7] Associado a cada *iterator* existe um registro de estado. Não existe variáveis estáticas e o algoritmo pode ser usado múltiplas vezes na consulta provendo paralelismo. A interface Operator prove os métodos *open, next, close*

compiladores. Esse modelo é usado há mais de 30 anos pela indústria de bancos de dados e é mostrado na Figura 8 para fins de comparação com o código otimizado do Tugsten, que virá mais a frente.

```
class Filter(child: Operator,
    predicate: (Row => Boolean)) extends Operator {
  def next(): Row = {
    var current = child.next()
    while (current != null && !predicate(current)) {
      current = child.next()
    }
    return current
  }
}
```

Figura 8: Modelo de iterador Volcano para operação de Filtro. Muito flexível porém não consegue usar eficientemente as características de CPUs modernas e as respectivas arquiteturas de *cache*.

Na versão 2.2+, o *Spark* usa uma técnica chamada *Whole-stage codegen*, implementada em um componente bastante eficiente chamado Tungsten, que permite geração de código em tempo real a partir dos planos físicos gerados pelo *Catalyst* durante a fase de otimização física 7(b). Além disso, o Tungsten possui seu próprio gerenciador de memória que evita os problemas de eficiência do *garbage collector* original da JVM (por exemplo, uma *string* de 4 bytes em UTF-8 ocupa 48 bytes na JVM).

Os planos físicos de execução gerados pelo *Catalyst* podem ter expressões a serem avaliadas e isto envolve muito custo computacional, pois são repetidos para todas as tuplas da relação. A geração personalizada de *byte-code* evita estes problemas convertendo as expressões para *byte-code* otimizado, uma única vez. Com isso, foi possível adicionar mais de uma centena de funções não-nativas ao *SparkSQL* com suporte a geração automática de *byte-code*. São funções sobre datas, *strings* e números, tais como *GetJsonObject*, *Decode*, *Encode*, *InitCap*, *FromUTCTimestamp*, *MonthsBetween*, *SoundEx*, *SHA2*, *FormatNumber*, *GetMapValue* etc.

Além disso, o Tungsten é capaz de: *i)* promover a fusão de operadores em conjunto, de modo que o código gerado se pareça com o código otimizado à mão; *ii)* manter os dados em registradores, em vez de *cache*/memória; *iii)* permitir que os compiladores destrinchem *loops* e usem as instruções do tipo *Single Instruction Multiple Data* (SIMD), em vez de *branches*. Assim, o *Spark* funciona como um compilador que consegue traduzir um código de uma consulta SQL, como esta:

```
select count(*) from store_sales where ss_item_sk = 1000
```

em um *byte-code* para um simples *loop* como este:

```
var count = 0
for (ss_item_sk in store_sales)
  if (ss_item_sk == 1000) count++
```

Este código pode receber o dado como um *stream* de tuplas e pode ser muito eficiente em processadores modernos graças ao JIT (*Just-in-time Compiler*) da JVM. Este técnica é baseada no trabalho de Neumann [2011] e o código gerado é muito mais simples que aquele mostrado na Figura 8.

Não se pode deixar de citar o fato que o Tungsten permite trabalhar com formato colunar para os dados na memória, melhorando muito o desempenho quando são processados dados do *HBase*, *Parquet* e outros formatos colunares de armazenamento de dados providos por *DataSources* do *Spark*. Este tipo de otimização física não faz parte do escopo desse trabalho, porém é ilustrado aqui para entendimento de todo o processo de otimização em uma visão holística.

Em resumo, pode-se dizer que o processo de otimização do *Spark* pode começar no *parser* do comando a ser executado, passando pela análise lógica do plano, sua otimização, geração do plano físico e terminando no Tungsten com geração de *byte-code* otimizado para os processadores modernos. É importante dizer que todo o processo de otimização se dá no *SparkDriver* para que o *Spark* possa distribuir o processamento entre os *workers*. Todos os IDs de atributos, funções, etc. são criados usando *AtomicLong* na JVM do *Driver* e são únicos por serem gerados na mesma JVM em modo *thread-safe*. A medida que o processo de otimização vai resolvendo os atributos, funções, etc, os objetos que representam estes elementos na *abstract syntax tree* (AST) são identificados e eles ganham o atributo numérico único que os acompanha até a geração de código para a JVM. O código gerado é então serializado e enviado aos *workers* correspondentes.

1.2 Sistemas de Gerência de *Workflow*

O termo *workflow* geralmente se refere ao processo de execução tanto de tarefas automatizadas, quanto manuais (com interação humana), integradas em um mesmo fluxo, o qual tem por objetivo a entrega de um determinado resultado. Para ilustrar a funcionalidade de um *workflow*, será apresentado o exemplo de um fluxo de processos da

fiscalização de ICMS em uma Secretaria de Fazenda Estadual. O resultado a ser obtido é o aumento da arrecadação encontrando os principais sonegadores e aplicando as cobranças de impostos e as respectivas multas por descumprimento da legislação. Trata-se de um processo complexo que começa com mineração de dados para descobrir as inconsistências entre informações declaradas nas notas fiscais de compra e de venda de uma dada empresa. Esta mineração pode ser feita automaticamente por *scripts* (previamente construídos) que acessam um *data warehouse* da Secretaria de Fazenda. Ao identificar os potenciais sonegadores, segue-se o processo distribuindo eletronicamente, aos fiscais, os relatórios com as discrepâncias. Neste momento, um processo manual onde os fiscais preparam a notificação a ser enviada ao contribuinte, é realizado. A notificação é adicionada ao sistema e o *workflow* volta a um processo automatizado composto por diversas tarefas de sistema. Quando o contribuinte comparece ao posto da Secretaria de Fazenda tem-se novamente um atendimento por parte de uma pessoa em um processo manual, porém suportado pelo sistema. Novos dados são inseridos eletronicamente neste sistema e o *workflow* volta a ser automatizado. Este é um caso típico de *workflow* complexo que pode ser gerenciado por um Sistemas de Gerenciamento de *Workflow* (SGW)

Como o problema de gerenciar *workflow* é bastante comum surgiu a necessidade de se criar uma organização não-governamental, a *Workflow Management Coalition* (WfMC), que cuidasse da padronização, distribuição de conhecimento e divulgação de soluções existentes neste setor. Segundo a WfMC, um *workflow* é a sistematização do processo de negócio, na sua totalidade ou em parte, onde documentos, informações ou tarefas são passadas de um participante a outro para execução de uma ação, de acordo com um conjunto de regras de procedimentos. Nesta definição, o participante pode se referir a um indivíduo ou a um processo automatizado em um sistema. Vale lembrar também que “processo de negócio” pode se referir a um experimento científico *in-silico*.

A WfMC tem se esforçado para padronizar uma linguagem de especificação de *workflow*, porém os principais fornecedores de SGW utilizam, sua própria linguagem além de conceitos baseados em diferentes paradigmas. A falta de consenso se deve, não só a interesses comerciais diversos, como também a ausência de uma teoria organizacional universal em padronização de conceitos de modelagem de processos de negócio [Van Der Aalst and Ter Hofstede, 2005].

Van Der Aalst and Ter Hofstede [2005] criaram uma linguagem de *workflow* e um soft-

ware chamado YAWL (*Yet Another Workflow Language*), que é um sistema de BPM/*Workflow* baseado em uma linguagem de modelagem concisa, que lida com transformações de dados e que se integra com recursos organizacionais e serviços WEB externos. Esta linguagem é proposta na perspectiva do fluxo de controle (*control-flow*) adequada para representar o exemplo de fiscalização de ICMS, citado anteriormente, que envolve tanto análise de dados quanto funções transacionais como emissão de auto de infração e de notificação fiscal.

Existe, porém, um conjunto grande de *workflow* centrado nos dados (*data-centric*) usado em análise de dados, onde as dependências entre estes e os processos intermediários definem um fluxo que pode ser alterado sem que a semântica da consulta seja modificada, abrindo oportunidades de otimização algébrica. Este tipo de *workflow* é chamado *dataflow* e usa álgebra relacional como base teórica para processar otimizações envolvendo relações. Em um caso mais geral, para tratar dados não estruturados ou semiestruturados, é necessário usar UDF, pois estas funções podem usar transformações arbitrárias nos dados ampliando imensamente a capacidade de processamento do *workflow* em termos de funcionalidades. Um exemplo possível é um *workflow* que lê dados textuais na WEB reconhecendo os títulos dos documentos a partir da tag `<title>` e identificando parágrafos pela tag `<p>` e `<div>`.

Uma UDF pode ser responsável por obter as URLs alvo fazendo o processo de rastreamento da rede (*WEB crawler*), filtrando e armazenando o conteúdo localmente em um formato adequado (*parquet*, por exemplo) e outra UDF pode ser responsável por ler este conteúdo e fazer a análise léxica e sintática do texto, preparando o dado para mineração de texto. Isso pode ser realizado por uma terceira UDF, enquanto uma quarta UDF pode aplicar um treinamento de um modelo de aprendizado de máquina. Nesta situação, pode-se integrar um processo complexo em um *workflow* totalmente automatizado, ainda que envolva intervenções manuais. A abordagem de *workflow* garante maior robustez e melhor resiliência ao processo.

É possível integrar um *dataflow* com gerenciadores de filas assíncronas de propósito geral (*general-purpose message broker*), tal como Apache MQ ou Amazon SQS (*Simple Queue Service*), que pode enfileirar transações em um sistema corporativo permitindo processamento transacional como efeito colateral. Entretanto, neste trabalho, este tipo de integração não será considerado devido ao desvio do assunto principal que é a análise de dados.

1.3 UDF

UDF são funções definidas pelo usuário e já são utilizadas em DSL relacionais tradicionais desde a década de 1990 para diversos propósitos. UDF podem ser classificadas em três tipos: *i)* UDF escalares; *ii)* UDF de agregação (*aggregate functions*) e *iii)* UDF de tabela (*table functions*) [Hsu et al., 2010]. A Tabela 2 mostra um exemplo para cada tipo. Neste trabalho, usa-se apenas UDF dos tipos escalar e de agregação.

Exemplo de Consulta	tipo de UDF
$SELECT\ myUDF(r.a)\ FROM\ r\ WHERE\ otherUDF(r.b)$	UDF escalar
$SELECT\ x, theUDF(r.c)\ FROM\ r\ GROUP\ BY\ x$	UDF de agregação
$SELECT * FROM\ myTableUDF(r.b)$	UDF de tabela

Tabela 2: Tipos de UDF

Nos SGBD tradicionais, costuma-se usar UDF escritas em C, C++, R, Python e Java, dependendo da implementação. No contexto deste trabalho, as UDF poderão ser escritas em Java, Scala ou Kotlin, mas estas UDF poderão invocar códigos externos à JVM escritos em outras linguagens, tais como C, C++, Fortran, R ou Python, ou mesmo serviços WEB (*WEB Services*). Entretanto, deve ser alertado que um *crash* em qualquer programa nativo externo à JVM pode derrubar o *Spark* e deve ser considerado uma interface baseada em rede ou comunicação entre processos como alternativa.

UDF tem atraído a atenção de pesquisadores como Rheinländer et al. [2017] que dedicam um estudo extenso ao papel de UDF nos *workflows* centrado nos dados. Eles executam os *workflows* em uma dada infraestrutura típica de *cluster* de computadores em uma organização sem compartilhamento (*shared-nothing*) usando o modelo *MapReduce* [Rheinländer et al., 2017]. Neste trabalho, os autores confirmam que UDF são estudadas em pesquisas na área de banco de dados nas últimas décadas e que linguagens declarativas, tal como SQL, inspiraram o desenvolvimento dos *frameworks* de *bigdata*, tal como o *Hadoop*, o Flink e o *Spark*. Todo o conhecimento sobre UDF, fruto destas pesquisas, está disponível para uso nestes *frameworks*, mas ainda existem questões em aberto.

Como ilustração, Rheinländer et al. [2017] criaram um *workflow* para uma consulta que analisa dois grandes grupos de artigos da Wikipédia reunidos em diferentes momentos (*i.e.* em diferentes janelas de tempo) para determinar as empresas que faliram em um determinado momento e são listadas na NASDAQ, juntamente com notícias sobre investigação de pessoas relacionadas a essas mesmas empresas. Trata-se de um *work-*

flow relacionado à gestão de conhecimento que usa dados semiestruturados obtidos da Internet. Este exemplo ilustra que os benefícios da otimização em UDF podem ser obtidos não apenas em dados estruturados, (por exemplo, os dados obtidos de bancos de dados relacionais), como também em dados obtidos de outras fontes não estruturadas como blogs, site de notícias, enciclopédias digitais e sites de relacionamento. Isso indica que o uso de UDF com operadores, como estes fazendo o papel de funções de segunda-ordem, permite aplicar regras algébricas e usar um formalismo matemático robusto. Este formalismo é uma extensão da álgebra relacional que vem sendo fruto de pesquisa nas últimas décadas.

Rheinländer et al. [2017] abordam a importância de encontrar uma solução para o problema de otimização desses *dataflows* e os principais desafios relacionados, além de classificar as abordagens possíveis para solução. Dentre os desafios citados está a dificuldade de estabelecer semântica para UDF uma vez que são opacas para o *framework* de execução no ambiente distribuído (como o *Spark*).

Vê-se, então, a importância de UDF em ambiente DISC para executar processos arbitrários associados a tuplas de relações. As UDF mais difundidas são do tipo *Statless* ou *Stateless scalar function* que processam apenas uma tupla por vez e não guardam estado entre o processamento de mais de uma tupla [8]. Outro tipo de UDF é a *Stateful* que permite processar mais de uma tupla guardando o estado entre o processamento delas. Um bom exemplo de UDF *Stateful* é uma para calcular média móvel (*moving average*) em uma série temporal. Dada uma janela de tempo de tamanho `w` é necessário guardar em um *buffer* os valores de `N` tuplas a medida que se calcula a média móvel [9]. UDF *Stateful* permite agregação de dados arbitrários usando funções definidas por usuário.

Agregação de dados é o ato de coletar uma quantidade de dados e resumir de acordo com certos critérios. Esta técnica constitue uma das bases da análise em *bigdata*, pois traduz dados brutos em informações de caráter analítico. Por exemplo, quando se observa nos jornais os resultados de uma eleição, distribuídas nos milhares de municípios do país, e organizadas por partido político, vê-se informações relacionadas e categorizadas, em vez de detalhes sobre cada voto. Isto é um exemplo de agregação de dados.

[8] Apesar de não guardar estado entre o processamento de mais de uma tupla, a UDF *Statless* pode produzir um efeito colateral afetando outra parte do domínio

[9] Veja exemplo de UDF *Stateful* implementada no *Hadoop* via Hive para uso com Oracle que está disponível no GitHub - `https://github.com/dwmclary/hive_udfs/blob/master/src/java/com/oracle/hadoop/hive/ql/udf/generic/UDFSimpleMovingAverage.java`

Na agregação de dados, informa-se ao sistema uma chave de agrupamento e uma função de agregação tal como soma (*sum*), média (*avg*), contagem (*count*), desvio-padrão (*stddev*), curtose (*kurtosis*) etc.

No Apache Spark é possível criar as chamadas UDAF (*User defined Aggregate Function*) que são funções de usuário capazes de guardar estado (*Stateful*). O armazenamento de estado ocorre entre as leituras das tuplas, de acordo com uma expressão de agregação, tal como uma operação análoga ao GROUP-BY da linguagem SQL. A vantagem desse tipo especial de UDF é permitir a escrita de código arbitrário de agregação em escalas lineares ou logarítmicas. A função de agregação pode atender a qualquer requisito específico do problema. O *Apache Spark* operacionaliza esta funcionalidade pois mantém um *buffer* de agregação único que armazena os resultados intermediários, para cada grupo de dados de entrada composto de um conjunto de tuplas que atendem ao critério de agrupamento. Para facilitar o desenvolvimento o *Spark* disponibiliza uma classe chamada *UserDefinedAggregateFunction* que pode ser estendida, implementando alguns métodos cujas assinaturas são bem definidas. Graças a Inversão de Controle (*IoC* - Inversion of Control) [10], o *Spark* é capaz de passar aos métodos o *buffer* atual e a tupla sendo processada permitindo a interação em alguns pontos do ciclo de vida do cálculo do agrupamento. Estes pontos são: *i)* inicialização; *ii)* atualização do *buffer* com os dados da tupla atual; *iii) merge* dos dados do *buffer* com os dados da tupla atual e *iv)* avaliação do dado no Buffer relacionado a tupla atual. Com a possibilidade de programar código arbitrário nestes *entrypoints* descritos, tem-se bastante liberdade para criar *Untyped User-Defined Aggregate Functions* (UDAF) relativamente sofisticadas, e no neste caso, estas funcionalidades podem ser escritas em Java, Kotlin, Scala, Python e R abrindo um sem-número de possibilidades.

Na próxima Seção, descreve-se a álgebra relacional que forma a base teórica para álgebra de *workflow* descrita mais adiante.

1.4 Álgebra Relacional

A Álgebra Relacional trata do formalismo para construção de consultas na área de SGBD. Graças a ela é possível formalizar operações, converter as consultas para uma estrutura de dados adequada e aplicar transformações sobre esta estrutura para otimizar

[10] Inversão de controle é o nome dado ao padrão de desenvolvimento de software onde a sequência de chamadas dos métodos é invertida em relação à programação tradicional. O controle é delegado a uma infraestrutura de software tal como um contêiner.

as consultas. As escolhas das transformações são realizadas a partir do conhecimento sobre as regras algébricas e sobre os métodos de pesquisa usados na seleção de registros (tuplas) de uma tabela. Estas seleções envolvem a definição dos índices e algumas características das tabelas (cardinalidade, seletividade, etc) [11]. A seletividade, por sua vez, depende da condição, que no caso mais geral, pode ser composta por combinações arbitrárias de expressões com condições conjuntivas e disjuntivas. As condições conjuntivas são amigas da otimização pois ligam operandos lógicos com operadores AND (representados pelo símbolo $\bigwedge$). Dessa forma, cada operando aplicado sobre os dados restringe um pouco mais a quantidade de registros na saída, fazendo com que a seletividade fique cada vez menor. Quanto menor a seletividade maior é a oportunidade de otimização pois permite que relações intermediárias possam ficar inteiramente na memória volátil aumentando o desempenho e a eficiência do sistema. Entretanto, os predicados podem conter condições disjuntivas que são os casos onde dois operandos lógicos são ligados por um operador OR (representados pelo símbolo $\bigvee$). Estas condições exercem um efeito inverso, aumentando a quantidade de registros na saída e, portanto, aumentando o valor da seletividade e diminuindo as oportunidades de otimização. As propriedades da álgebra Booleana ajudam a fatorar o predicado reorganizando os operadores AND e OR de forma a facilitar o processo de otimização da consulta.

Um outro aspecto relacionado ao desempenho dos SGBD é a escolha do método de execução da consulta. Existem tipicamente dois métodos: *i)* execução com materialização de resultados intermediários e *ii)* execução em *pipelining* ou processamento baseado em fluxo (*stream-based processing*).

A execução com materialização de resultados é a solução mais simples e óbvia, porém, para relações muito grandes, pode penalizar o desempenho do processamento pois exige uma gravação das relações intermediárias entre cada operação, e uma nova leitura desta mesma relação intermediária posteriormente, que via de regra impõe um alto custo computacional de entrada e saída *I/O*.

Por outro lado a execução das operações algébricas em *pipelining*, pode parecer mais complexa, mas evita que se materialize todas as relações intermediarias, valendo-se de estruturas de *cache* e mantendo os resultados intermediários na memória.

A Figura 9 mostra uma execução em *pipelining* de uma cadeia de processamento

[11] A cardinalidade mede a quantidade total de registros e a seletividade mede o quanto uma dada condição (predicado) reduz a relação de saída quando se aplica o operador SELEÇÃO $\sigma_{predicado}(R)$ da álgebra relacional.

com três operadores relacionais aplicados sobre uma relação R. Neste modelo a execução da primeira operação da cadeia (*oper1*) sobre a primeira tupla (*tupla1*) da relação de entrada gera um resultado que pode ser passado imediatamente para a segunda operação da cadeia de processamento (*oper2*) liberando a primeira operação que pode, em paralelo, ler e processar a segunda tupla da relação de entrada (*tupla2*). Este processo se repete e após o processamento de quatro tuplas tem-se a configuração que pode ser visto na Figura 9. Vale ressaltar que na configuração mostrada, a primeira tupla da relação de entrada R foi processada pelos três operadores em *pipeline* e já está disponível na relação de saída T e concomitantemente o operador *oper3* está processando a *tupla2* em paralelo com o *oper2* que processa a *tupla3*, que por sua vez está dependendo do operador *oper1* que executa sobre a *tupla4*, mas já executou seu processo sobre as três primeiras (*tupla1*, *tupla2* e *tupla3*).

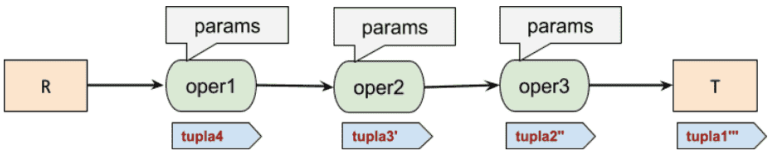


Figura 9: Execução em *pipelining* que pode usar o modo de execução *source* ou o modo de execução *sink*.

Este modelo de *pipelining* é conhecido como *source* em Engenharia de Software, onde se estuda o padrão de projeto Filtro (*design pattern Filter*), pois a fonte de dado *source* empurra dado na *pipeline*. O outro modelo de *pipelining* é o *sink* onde o alvo (*target*) suga o dado quando precisa.

O que determina o modelo de *pipelining* é a implementação, mas ambos produzem o mesmo resultado. As estratégias *source* ou *sink* são escolhidas de acordo com os requisitos funcionais e não funcionais da aplicação, por exemplo, um requisito de carregamento tardio (*lazy loading*) pode exigir a estratégia *sink* [12].

Antes de executar a consulta o SGBD precisa otimizar a árvore de consulta. Esta árvore é criada a partir da análise gramatical do comando SQL, fornecido pelo usuário. Considere, por exemplo, o comando SQL abaixo descrito em Silberschatz et al. [2011]:

$$SELECT\ I.name, C.title\ FROM\ instructor\ I, teaches\ T, course\ C$$
$$WHERE\ C.course\_id = T.course\_id$$

[12] Funções de segunda ordem da programação funcional facilitam a implementação da estratégia *sink*

$AND\ T.instructor\_id = I.id$

$AND\ I.depto\_name = 'Music'$

Este comando SQL pode ser representado em uma arvore de consulta [13], como a da Figura 10, no seu formato original. Esta árvore representa a consulta que retorna os nomes de todos os instrutores do departamento de música, juntamente com todos os nomes de curso onde estes instrutores lecionam.

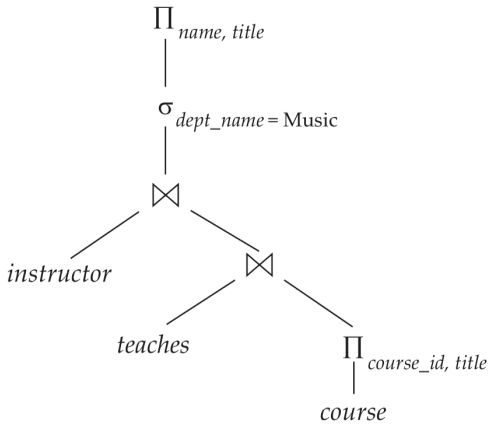


Figura 10: Árvore de consulta para um comando SQL específico.

A árvore mostrada na Figura 10 pode sofrer várias transformações como fruto do processo de otimização, onde se usam heurísticas para escolher as melhores estratégias, que por sua vez dependem das informações de cardinalidade e seletividade, já mencionadas anteriormente. Além disso, as heurísticas de otimização dependem das regras de transformação da álgebra relacional, pois elas garantem que não há modificação na semântica da consulta original, já que são baseadas nas propriedades da teoria de conjuntos. A seguir, aparece uma lista de 12 regras gerais mencionadas na Seção 1.1.1 (Os detalhes da API de otimização do *SparkSQL* e do *Catalyst*) onde foi introduzido o conceito de otimização.

Regras gerais de transformação da Álgebra Relacional

A lista de regras de transformação da Álgebra Relacional apresentadas no Anexo

[13] Uma estrutura de dados mais neutra para representar uma consulta é o grafo de consulta (*query graph*) [Elmasri and Navathe, 2015].

A trata apenas daquelas associadas diretamente a otimização de filtros contíguos com UDF, no escopo da avaliação experimental realizada.

Além das regras mostradas no Anexo A é possível realizar diversas outras transformações usando também as regras da Álgebra Booleana (leis de De Morgan) e outras regras da teoria de conjuntos. O importante é observar que as heurísticas de otimização compõem diversas destas regras em sequência com o objetivo de minimizar a seletividade e a cardinalidade das relações intermediárias para diminuir o custo computacional de acesso ao dado e o processamento da consulta.

Com as regras de transformação da Álgebra Relacional estabelecidas já é possível apresentar a Álgebra de *Workflow* que introduz UDF no contexto destas propriedades mostradas anteriormente. Na próxima Seção, descreve-se a álgebra de *workflow* que se apresenta como uma das soluções para o problema de estabelecer semântica para UDF, e propõe heurísticas para otimização de consultas usando regras algébricas semelhantes a estas descritas nesta seção.

1.5 Álgebra de *Workflow*

Rheinländer et al. [2017] destacam a relevância da otimização de UDF e apontam as abordagens algébricas para execução de *workflow* de Ogasawara [2011] como uma das soluções para a anotação de semântica em UDF, que permite otimizações. Esta proposta algébrica tem como premissa a existência de operadores que regem a execução das atividades e que impõem uma semântica sobre a produção e o consumo dos dados. Esses operadores podem ser considerados como adicionais a álgebra relacional com a característica de associarem as relações de entrada a uma UDF escrita em uma linguagem de programação disponível. Estes operadores funcionam como anotadores de semântica nas UDF para que o otimizador use as heurísticas adequadas em suas regras.

Na álgebra de *workflows* de Ogasawara et al. [2011], os dados são uniformemente representados por meio de relações e as atividades são regidas por operações algébricas que possuem semântica sobre a produção e o consumo dos dados. Considera-se, nesta álgebra, que atividades consomem e produzem relações. Isso traz uma uniformidade no modelo de dados e de processo, possibilitando a geração de *workflows* prontos para execução em paralelo a partir de uma definição independente do modelo de computação (premissa da MDA). As relações são definidas como um conjunto de tuplas de

dados primitivos (*i.e.*, inteiro, real, alfanumérico, data, *timestamp*, *blob*, etc) ou referência a arquivo (via URI: *Uniform Resource Identifier*). No que tange ao tratamento de *blob* e arquivos, o seu formato é considerado opaco. Desta forma, tem-se uma uniformização do tratamento dos arquivos, sejam eles textuais, semiestruturados ou binários. Assim, é de responsabilidade das atividades, mais especificamente das UDF, no *workflow* saber consumir ou produzir estes dados [Ogasawara et al., 2011].

Cada relação R possui um esquema $\mathcal{R}$ e pode ser especificada como $R(\mathcal{R})$. Dada uma relação $R(\mathcal{R})$, representa-se $atr(R)$ como um conjunto de atributos de R e $key(R)$ como o conjunto contendo os atributos chave de R. De modo análogo à álgebra relacional, relações podem ser manipuladas por operações de conjunto: união ($\cup$), interseção ($\cap$) e diferença ($-$), desde que os seus esquemas sejam compatíveis (aridade da relação e domínio de cada atributo). Pode-se atribuir uma relação a variáveis de relação para posterior reuso usando a atribuição $\leftarrow$ (ex.: $T \leftarrow R_1 \cup R_2$) [Elmasri and Navathe, 2015].

No escopo deste trabalho, uma atividade compreende uma UDF (encapsulando a invocação de um programa ou a execução de uma expressão da álgebra relacional, tal como uma operação de agregação) e esquemas de relações, tanto de entrada quanto de saída. As atividades do *workflow* são regidas por operações algébricas que especificam a razão de consumo e produção entre as tuplas. Esta característica possibilita um tratamento uniforme para as atividades viabilizando a realização de transformações algébricas. A álgebra inclui seis operações (resumidas na Tabela 3): Map, $SplitMap$, $Reduce$, $Filter$, $SRQuery$ e $MRQuery$. A maioria das operações algébricas consome uma única relação, com exceção da operação $MRQuery$ que consome uma sequência de relações. As primeiras quatro operações são usadas para apoiar atividades que executam programas encapsulados por meio de UDF. As duas últimas operações são usadas para executar atividades que processam expressões de álgebra relacional. Isso significa que a UDF deve ser compatível com a operação da atividade em termos de interface para consumo e produção de dados [Hsu et al., 2010].

Tabela 3: Resumo das operações algébricas

Operação	UDF	Operandos adicionais	Resultado	Cons. $\times$ Prod. tuplas
Map	programa	Relação R	Relação S	$1:1\ por\ \|R\|$
SplitMap	programa	Relação R	Relação S	$1:m\ por\ \|R\|$
Reduce	programa	Relação R e atrs	Relação S	$n:1\ por\ \|\pi_{atrs}\|$
Filter	programa	Relação R	Relação S	$1:(0-1)\ por\ \|R\|$
SRQuery	exp. relacional	Relação R	Relação S	$n:m$
MRQuery	exp. relacional	Relações $(R_1 \cdots R_i)$	Relação S	$(n_1 \cdots n_i):m$

Neste contexto, os *workflows* podem ser otimizados para execução paralela por meio de transformações algébricas. Cada transformação aplicada, apesar de garantir que o *workflow* produza o mesmo resultado, traz uma diferença em termos de custo computacional. Em outras palavras, expressões algébricas equivalentes produzem diferentes planos de execução do *workflow*. Pode-se avaliar o custo destes planos por meio de uma função de custo. Desta forma, a abordagem algébrica possibilita a visualização do problema de execução paralela de *workflow* de modo análogo à otimização de consultas em bancos de dados relacionais.

Assim, notação $A_o < R_1, ...R_n, T, F >$ representa uma atividade A regida por um operador algébrico o que depende das relações de entrada R_1, ... R_n, de uma função definida pelo usuário F e que gera uma relação de saída com esquema T. Cada uma das relações de entrada está associada aos esquemas: S_1, ... S_n, respectivamente. Como foi visto, a álgebra de *workflow* [Ogasawara et al., 2011] descreve seis operadores, dos quais dois são detalhados neste trabalho: (*wMap* e *wFilter*). O operador *wMap* é definido da seguinte forma: $T \leftarrow wMap(A, R)$, onde a atividade A produz uma única tupla na relação de saída T para cada tupla consumida na relação de entrada R. Os esquemas de R e T podem ser diferentes em um caso mais geral. Já o operador *wFilter* é descrito assim: $T \leftarrow wFilter(A, R)$, onde a atividade A produz uma ou nenhuma tupla na relação de saída T para cada tupla consumida na relação de entrada R. Neste processo, o esquema de T (saída) é o mesmo de R (entrada). Portanto, é possível observar que a semântica dos operadores *wMap* e *wFilter* [Ogasawara et al., 2011] é compatível com seus análogos *map* e *filter* do *SparkSQL*.

Na Seção 1.4 foram listadas as 12 regras gerais da álgebra relacional consolidadas e apresentadas por Elmasri and Navathe [2015]. Pode-se, então, estender estas regras na Álgebra de *Workflow* para contemplar UDF e com isso permitir que *Dataflow* possa ser otimizado mesmo quando esteja executando atividades extranas escritas em Python ou R. Esta abordagem é descrita na seção 3.6.

Como já foi mencionado anteriormente, para otimizar um *Dataflow* contendo UDF é necessário colher dados de proveniência retrospectiva para enriquecer o catálogo com informações de custo de execução dessas UDF e apoiar as heurísticas com estas informações adicionais. Nesse sentido aborda-se a seguir o tema proveniência.

1.6 Proveniência

Segundo a *World Wide Web Consortium* (W3C) em sua especificação do modelo de dados para proveniência (PROV-DM), essa é definida como: “um registro que descreve as pessoas, instituições, entidades e atividades envolvidas na produção, influência ou entrega de um dado ou coisa. Em particular, a proveniência da informação é crucial para decidir se ela é confiável, como deve ser integrada a outras fontes de informações diversas e como dar crédito a seus criadores ao reutilizá-la.”. A preocupação principal da W3C é a confiabilidade da informação encontrada na WEB, por ser este um ambiente aberto e inclusivo, onde usuários encontram informações muitas vezes contraditórias ou questionáveis e a proveniência pode ajudar no julgamento sobre a veracidade da informação. Na definição, observa-se a necessidade de se registrar quem criou ou alterou o dado ou o processo. No contexto deste trabalho, a confiabilidade e a responsabilidade são muito importantes, mas existem outras facetas em que a proveniência é útil. A lista das áreas tratadas pela proveniência são : *i)* origem da responsabilidade por uma dada informação; *ii)* registro de origem da informação ; *iii)* registro de evolução e versionamento da informação e do processo; *iv)* justificativa para as decisões a respeito dos processos; *v)* implicação, indicando quais dados levam a um dado resultado ; *vi)* administração do acesso e segurança da informação. Vale ressaltar que esse assunto é amplo e complexo e, portanto, sofrerá um recorte no escopo deste trabalho.

Além do W3C, uma outra iniciativa de especificação de proveniência é do *Open Provenance Model* (OPM). O objetivo principal do OPM é apoiar a avaliação de vários atributos dos dados, tais como confiabilidade, precisão e pontualidade, e para tal, define um modelo de grafos para proveniência que descreve as arestas como a relação entre as ocorrências, que são representadas pelos nós (dos grafos). Estes nós podem ser: *i)* Artefatos que são partes de dados de valor fixo e contexto que possivelmente representam uma entidade em um determinado estado; *ii)* Processos que são executados em artefatos para produzir outros artefatos e *iii)* Agentes que indicam as entidades que estão controlando o processo, tal como um usuário. Arestas também podem ter anotações para fornecer as informações sobre como uma ocorrência tem efeito causal sobre outra;

Um dos objetivos da proveniência é permitir a reprodutibilidade de experimentos *in-silico*, pois com isso outros pesquisadores podem verificar os resultados reexecutando os experimentos em seu próprio ambiente. Chirigati et al. [2017] apresentam duas soluções

promissoras para reprodutibilidade. Uma delas é o ReproZip [14], projetada e implementada pela comunidade científica, em particular, pela equipe da Universidade de Nova York. Outra ferramenta é o Docker [15] que implementa virtualização leve e permite que se defina em um *script* no formato texto, um ambiente completo de *runtime* que reproduza um experimento. Assim, autores e revisores podem colaborar entre si. O ReproZip tem a vantagem de rastrear automaticamente as dependências de seus experimentos existentes (escritos em *scripts* Python) e criar um pacote independente. O Docker complementa o ReproZip uma vez que é capaz de criar um ambiente de virtualização para o experimento de forma leve e fácil de usar e implantar.

Como foi mencionado na Introdução, o *Spark* permite que se obtenha métricas e informações sobre o ambiente de execução via a API *SparkListener* que é um mecanismo para interceptar eventos do escalonador do *Spark*. Estes eventos são emitidos durante a execução de uma aplicação *Spark*, e isto significa que se pode filtrar e processar informações via *SparkListener* e armazenar tais informações como proveniência retrospectiva em um repositório de apoio [16]. Informações da quantidade de *threads* e de memória de acesso randômico utilizada por um processo também podem ser coletadas e armazenadas neste mesmo repositório, para apoiar otimização relacionada à materialização de resultados intermediários.

Ainda que existam outros modelos de proveniência, optou-se neste trabalho limitar a pesquisa aos modelos descritos devido ao escopo limitado a otimização de operadores *wFilter*. Outros modelos poderão ser motivo de investigação em trabalhos futuros.

Para encerrar este capítulo é necessário discorrer sobre o modelo arquitetônico, que serviu de inspiração para a implementação do *framework* WfF devido aos requisitos específicos de independência de plataforma, e isso é o tema da próxima Seção.

1.7 Arquitetura dirigida a modelos

A MDA foi proposta pelo OMG (*Object Management Group*), em 2001, e é apresentada como um novo paradigma de desenvolvimento de software. A arquitetura dirigida a modelos concentra-se nos estágios de projeto e implementação do processo de desenvolvimento de software, e não trata de engenharia de requisitos baseada em modelos,

[14] Experimentos definidos em linguagem Python podem usar o ReproZip - `https://www.reprozip.org` para empacotar o código principal e todas as dependências, incluindo arquivos de dados e binários utilizados pelo experimento.

[15] Veja o site `https://docker.com`.

[16] Santos [2018] desenvolveu um trabalho sobre coleta de proveniência no Apache Spark

nem nos testes baseados em modelos. No entanto, as bases que formam a MDA são suficientes para atingir os objetivos de independência entre requisitos do *dataflow* e a plataforma de execução, que é um dos objetivos do *framework* WfF proposto.

Na especificação da OMG, a MDA usa um subconjunto de modelos da UML (*Unified Modeling Language*) [17] para descrever um sistema. Nesta abordagem, são criados modelos em diferentes níveis de abstração e a partir de um modelo independente de plataforma em alto nível é possível, em princípio, gerar um programa executável para uma dada plataforma, sem intervenção manual.

A Figura 11 mostra as três camadas da arquitetura e os processos associados às mesmas. A MDA recomenda a produção de três tipos de modelos abstratos de sistema que são: *i)* um modelo independente de computação - *computation independent model* (CIM); *ii)* um modelo independente de plataforma - *platform independent model* (PIM) e *iii)* modelos específicos de plataforma - *platform specific model* (PSM).

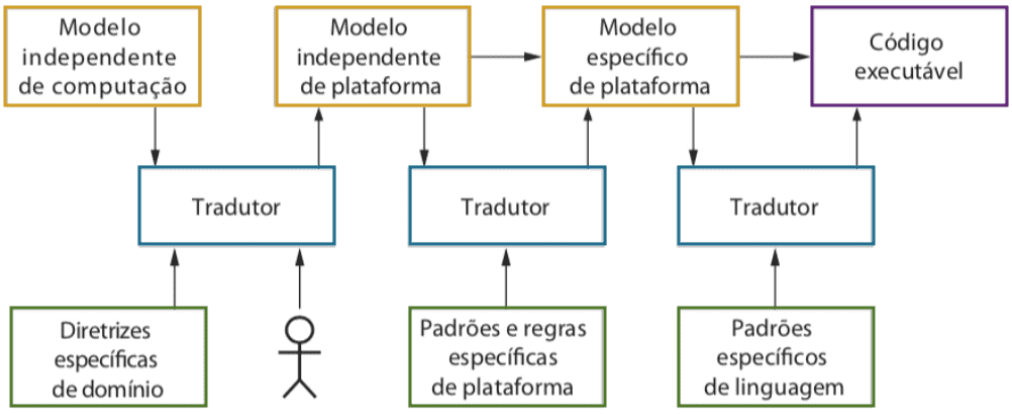


Figura 11: As três camadas da arquitetura MDA e os artefatos e processos associados (figura baseada em imagem no livro [Sommerville, 2011])

O CIM modela as importantes abstrações do domínio usado no sistema e são também chamados “modelos de domínio” pela sua característica de depender apenas dos requisitos funcionais. Em geral, o usuário pode desenvolver vários CIMs diferentes, refletindo diferentes visões do sistema. No caso dos *dataflows* dos cientistas de dados, a verificação de uma hipótese pode exigir a execução de vários experimentos em um certo domínio o que equivale a mais de um CIM, na nomenclatura da MDA.

O PIM serve ao propósito de modelar a operação do sistema sem referência a sua

[17] No *framework* WfF em vez de UML, como linguagem de especificação, usa-se um grafo acíclico direcionado construído a partir de um código de texto plano segundo a gramática de uma eDSL - veja a Seção 1.8 para detalhes

implementação. O PIM, como definido pela *Object Management Group* (OMG), é descrito por meio de modelos da *Unified Modeling Language* (UML) que mostram a estrutura estática do sistema e como ele responde a eventos externos e internos.

O PSM permitem transformar o modelo independente de plataforma com um PSM separado para cada plataforma de aplicação. Em princípio, pode haver camadas de PSM, onde cada uma acrescenta alguns detalhe específico. Assim, o PSM de primeiro nível pode ser um *middleware* específico, mas independente do banco de dados. Quando um banco de dados específico for escolhido, um PMS específico de dados pode ser gerado.

Para efeito deste trabalho, a MDA serviu apenas de inspiração e, portanto, a proposta de arquitetura não foi seguida a risca. Optou-se por uma solução mais simples usando as ideias principais de abstração e adaptabilidade e além disso, em vez de UML, optou-se por um grafo acíclico direcionado para especificar o PIM do *dataflow*.

A Seção 3.2 descreve como foi a abordagem usada para implementar o *framework* de forma que se possa obter os benefícios do modelo independente de computação (*CIM*). O CIM, é especificado pelo cientista de dados na forma de um texto plano seguindo uma gramática definida em uma eDSL. Este texto é compilado e traduzido para um PIM na forma de um grafo que é persistido. Na sequência gera-se o código Scala que pode ser submetido ao *Apache Spark* para execução no *cluster*.

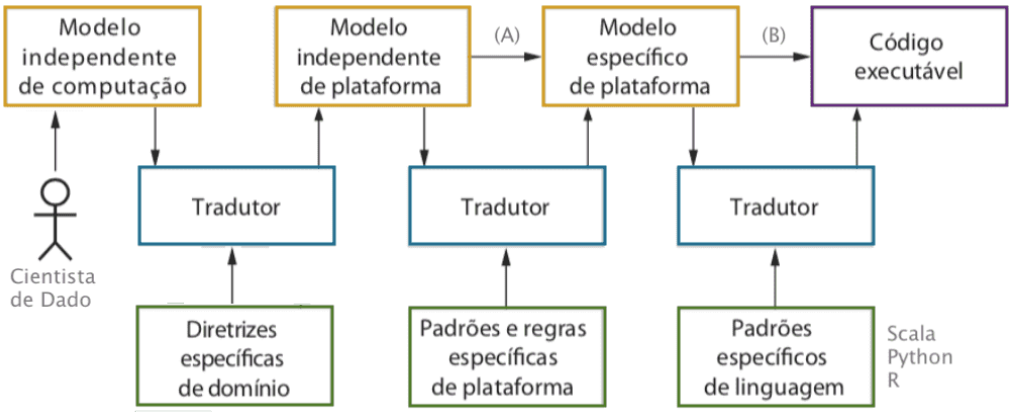


Figura 12: As três camadas da arquitetura MDA e os artefatos e processos associados adaptada para o contexto de analise de dados em ambiente distribuído do *framework* WfF.

A Figura 12 mostra as modificações na arquitetura MDA original para adequar à realidade específica do *framework* WfF. Na modificação proposta a única intervensão humana ocorre na especificação do *dataflow* pelo cientista de dado (no CIM) que também é responsável pela especificação da ligação (A) entre o PIM e o PSM e também a

ligação (B) entre o PSM e o código gerado. No entanto, é necessário destacar que isso é feito via comandos da eDSL que atuam em um alto nível de abstração, usando os *tokens* *Resource*, *Jobs*, *Execution* e suas composições, como pode ser visto nos fragmentos de programa das Figuras 25, 26 e 27. O modelo independente de computação (CIM) aparece na figura 24.

O cientista de dado pode contar com o apoio de um engenheiro de dados para ajudar com as especificações dos elementos *Resource*, *Jobs* e *Execution* da eDSL, quando estes exigirem informações específicas de plataforma que o mesmo desconheça. Em outras palavras, nos ambientes para analise de dados no contexto DISC, este modelo favorece a separação de responsabilidades (*Separation of Concerns*) entre os cientistas e os engenheiros de dados. Na Seção 3.2 apresenta-se uma descrição mais detalhada da Figura 12 no contexto do *framework* WfF.

Enquanto que no modelo de desenvolvimento Orientado a Objetos os programas são os artefatos de saída principais, na MDA, os modelos são as saídas principais do processo de desenvolvimento [Frankel, 2003], [Brown, 2004], [Singh and Sood, 2009] e [McClatchey, 2018]. Isto ocorre porque na MDA os programas são gerados automaticamente, a partir dos modelos. Isto aumenta o nível de abstração e, dessa forma, os usuários não precisam mais se preocupar com detalhes da linguagem de programação ou com as especificidades das plataformas de execução. Quando se trata de análise de *bigdata* em ambiente distribuído (DISC), a plataforma de execução é ainda mais complexa e a MDA produz mais benefícios pois os usuários são cientistas de dados, na sua maioria, com pouco conhecimento de engenharia de software. Nesta Seção descreve-se um pouco da arquitetura orientada a modelos e como ela foi usada no *framework* WfF para permitir a independência de plataforma por parte dos cientistas de dados, quando estes estiverem usando o *framework* proposto.

Na Seção seguinte detalha-se a eDSL definida, tendo em vista os requisitos funcionais impostos pela MDA além dos requisitos da execução de *dataflow* em ambiente distribuido, com opção de otimização usando a API do *Catalyst*.

1.8 Linguagem Específica de Domínio para *Workflow*

Nesta Seção aborda-se a importância de usar uma (DSL para a especificação do *Workflow*. O *framework* deve prover também o armazenamento, em repositório centralizado, de uma representação em formato independente de plataforma. Neste caso

foi usada uma representação na forma de Grafo Acíclico Direcionado - *Directed Acyclic Graph* (DAG), permitindo a execução do *Workflow*, pelo usuário. A definição de uma DSL para o *framework* fornece vários benefícios para o usuário quando comparada a uma API, e por esse motivo foi estudada a viabilidade de sua implementação.

Um *framework*, como o proposto neste trabalho, poderia expor uma API de acesso, e o usuário poderia usá-la para interagir com o sistema, o que é perfeitamente factível, já que uma API pode especificar relações entre classes implementadas no *framework*. Uma API, para ser útil, precisa ser definida de forma muito clara com boas escolhas para os nomes dos objetos, métodos e atributos. Nomes significativos melhoram a leitura do programa e os valores assumidos como padrão facilitam a utilização pelo usuário.

Ocorre, porém, que APIs exigem o conhecimento detalhado de classes e métodos internos à implementação e introduzem ruídos textuais no programa que não são essenciais à utilização das funcionalidades. Uma API exige código imperativo com utilização de laços (*loop*) e desvios (*branch*) tal como *if/then/else*. Por outro lado, uma DSL para especificação de *workflow* de análise de dados, pode permitir uma interface declarativa, o que simplifica muito a sua especificação por parte do usuário. O programa ganha também expressividade e clareza.

DSLs já são estudadas e utilizadas há várias décadas. O exemplo mais marcante é o da linguagem SQL que é específica para o domínio de gerência de banco de dados. Como ela pretende resolver um problema específico pode ser muito mais concisa que uma linguagem de uso geral. A linguagem SQL é excelente na tarefa de especificar os requisitos de acesso ao banco de dados, mas quando utilizada na prática em programas C++ ou Java, por exemplo, precisam ser inseridas como um corpo estranho, dentro dos programas. Este corpo estranho aparece, em geral, como *strings* de caracteres concatenados dificultando a compilação pela linguagem hospedeira. Em geral, a compilação do código SQL é delegada ao SGBD e as verificações sintáticas só podem ser feitas no SQL em tempo de execução. Este é um problema das DSLs clássicas.

Mais recentemente surgiu o conceito de eDSL (*Embedded DSL*), como construções regulares da linguagem hospedeira. Elas usam intensamente o padrão de projeto *Builder* definido por Gamma et al. [1994], e outras características sintáticas das linguagens hospedeiras, o que facilita a integração da DSL com a gramática padrão da linguagem. Linguagens como Haskell, Scala e Kotlin implementam estas facilidades. No caso do *framework* em pauta, inicialmente avaliou-se a implementação em linguagem Scala, po-

rém, com o surgimento da linguagem Kotlin, ficou claro que essa seria a melhor escolha para ser usada como linguagem hospedeira da DSL do *framework*. Um programa Kotlin é de fácil utilização em ambiente operacional de JVM que é o caso do *Apache Spark*. A linguagem Scala exige uma curva de aprendizado maior devido a certos problemas de projeto como os chamados *implicts*. Dito isso, lista-se a seguir, as características úteis da linguagem Kotlin para especificação de eDSL.

Uma eDSL possui uma estrutura própria associada à sua gramática e, portanto, pode ser chamada de linguagem, no contexto da programação. Como linguagem seria muito útil ter a possibilidade de escrever código da forma mais natural possível, como se fosse um texto em linguagem natural tal como o português ou o inglês. Em linguagens naturais tem-se os parágrafos, as sentenças e as palavras organizadas de forma a criar um sentido (semântica). Os posicionamentos das palavras indicam sua classe gramatical e o usuário interpreta o significado de forma natural. Para prover este tipo de funcionalidade na linguagem Kotlin, os projetistas da linguagem implementaram as seguintes funcionalidades que aparecem aqui com o nome original em inglês: *i) lambdas with receivers* - emprega uma estrutura de aninhamento muito útil em composições de objetos interdependentes; *ii) infix method operators* - objetos nomeados como parâmetros de operadores *infix* permitem escrever sentenças como em linguagem natural. ; *iii) functions extensions on primitive types* - a capacidade de adicionar métodos em tipos primitivos permite criar gramáticas para literais de forma legível e simples ; *iv) invoke convention* permite invocar objetos como se fossem funções e se os objetos são nomeados usando *tokens* da linguagem isso facilita ainda mais a definição de uma gramática simples e fluida.

A Seção 3.4 mostra um exemplo de código escrito usando a eDSL do *framework*. O objeto *Workflow* é composto de itens diversos, e procurou-se definir uma gramatica mais próxima possível da linguagem natural usando-se quatro características da linguagem Kotlin para *design* de eDSL citadas acima.

É importante observar que a linguagem Kotlin implementa inferência de tipos e é fortemente tipada, o que permite a verificação da especificação do *workflow* em tempo de edição do programa quando usa-se uma IDE (*Integrated Development Environment*) apropriada tal como o Eclipse e o IntelliJ Idea. Entretanto é possível usar outros editores como VI ou Sublime Text, por exemplo.

Em relação a gramática da linguagem WfF pode-se dizer que ela pode ser definida

em termos da forma estendida de Backus-Naur - *extended Backus-Naur form* (EBNF) [Pattis, 2013]. Esta modalidade é muito usada na ciência da computação como uma família de notações de metasintaxe para expressar uma gramática livre de contexto, sendo inclusive adotada pela *International Organization for Standardization* (ISO) na ISO/IEC 14977.

```
Expression ⇐ Expression , ( "&&" | "<" | "+" | "-" | "*" ) , Expression
           | Expression, "[", Expression, "]"
           | Expression, ".", "length"
           | Expression, ".", Identifier, "(", [ Expression { ",", Expression } ],
             ")"
           | IntegerLiteral
           | "true"
           | "false"
           | Identifier
           | "this"
           | "new", "int", "[", Expression, "]"
           | "new", Identifier ,"(" ,")"
           | "!", Expression
           | "(", Expression, ")"
           ;
Identifier is one or more letters, digits, and underscores, starting with a letter
IntegerLiteral is one or more decimal digits
EOF is a distinguished token returned by the scanner at end-of-file
```

Figura 13: Especificação de gramática para expressão envolvendo literais inteiros identificadores e diversos operadores lógicos e aritiméticos usando EBNF.

A EBNF é uma especificação simples composta de regras onde cada uma tem três partes: um lado esquerdo *left–hand side* (LHS), um lado direito *right–hand side* (RHS), e o caractere ⇐ separando esses dois lados. O simbolo ⇐ deve ser lido como: “está definido como”. O LHS é uma palavra escrita em minúsculas que nomeia a regra EBNF. O RHS fornece uma descrição desse nome. O carácter | permite especificar escolhas alternativas. Os caracteres [e] permitem especificar itens opcionais. Os caracteres { e } permitem especificar itens repetitivos. Os caracteres (e) permitem especificar agrupamentos de itens. Além disso, os símbolos terminais, conhecidos como *tokens* aparecem entre aspas. A EBNF define também o operador *is* que provê uma forma de descrever um item LHS em linguagem natural descritiva. Assim é possível especificar,

por exemplo, uma expressão usando EBNF como mostrado na Figura 13.

Partindo da definição EBNF para expressões, pode-se definir a gramática para uma calculadora simples, e ir estendendo a gramática até chegar a uma linguagem de programação tal como Java. Na Figura 14 apresenta-se a gramática para um subconjunto da linguagem Java [18].

```
Goal ⇐ MainClass, { ClassDeclaration }, EOF;
MainClass ⇐ "class", Identifier, "{", "public", "static", "void",
            "main", "(", "String", "[", "]", Identifier, ")", "{",
            Statement, "}", "}";
ClassDeclaration ⇐ "class", Identifier, [ "extends", Identifier ], "{",
                   { VarDeclaration }, { MethodDeclaration } "}";
VarDeclaration ⇐ Type, Identifier, ";";
MethodDeclaration ⇐ "public", Type, Identifier, "(", [ Type, Identifier, { ",", Type,
                    Identifier }, ], ")", "{", { VarDeclaration }, { Statement },
                    "return", Expression, ";", "}";
Type ⇐ "int", "[", "]"
     | "boolean"
     | "int"
     | Identifier
     ;
Statement ⇐ "{", { Statement }, "}"
          | "if", "(", Expression, ")", Statement, "else", Statement
          | "while", "(", Expression, ")", Statement
          | "System.out.println", "(" , Expression, ")", ";"
          | Identifier, "=", Expression, ";"
          | Identifier, "[", Expression, "]", "=", Expression, ";"
          ;
```

Figura 14: Especificação de gramática para um subconjunto da linguagem Java.

É importante ressaltar que não existe nenhuma recomendação sobre o uso de cores para identificar os elementos da metalinguagem EBNF e que o uso da cor vermelha nas Figuras citadas acima, foi uma escolha para favorecer o entendimento do conceito.

Outras metasintaxes são utilizadas, além da EBNF, mas em geral adaptadas desta [19]. Vê-se, portanto, que é de grande importância conhecer a definição da EBNF quando se deseja especificar uma gramática de linguagem tal como uma eDSL. Uma definição formal da gramática da eDSL do *framework* WfF aparece na Seção 3.5.

[18] Gramática para mini Java baseada nas notas de aula disponíveis em `http://cs.fit.edu/~ryan/cse5251/`

[19] Por exemplo, a linguagem Java é definida usando a metasintaxe disponível em *2.1. Context-Free Grammars* - `https://docs.oracle.com/javase/specs/jls/se10/html/jls-2.html`, que é uma variação da EBNF. A definição da gramática da linguagem Java em si, que usa esta metasintaxe pode ser vista em `https://docs.oracle.com/javase/specs/jls/se10/html/jls-19.html`

(Página intencionalmente deixada em branco)

Capítulo 2 Trabalhos Relacionados

Existem poucas abordagens algébricas para *workflows*, principalmente no que tange àquelas que apoiam a execução paralela das atividades representadas por UDF. Entretanto, há diversas abordagens cujo objetivo é a otimização da execução paralela de atividades em ambientes de dados em larga escala como mencionado por Ogasawara [2011]. Para avaliar os diferentes trabalhos relacionados a otimização de *dataflow* algébrico foi feita uma busca sistemática, datada de 09 de março de 2018, na base de dados do Scopus a partir das palavras-chave: `("dataflow"OR"workflow"OR"UDF") AND "optimization"AND("parallel execution"OR"distributed execution")`

A busca retornou 412 documentos distribuídos da seguinte forma: *i)* 76 estão relacionados a computação científica ; *ii)* 59 sobre *dataflow* ; *iii)* 11 estão relacionados a *MapReduce* ; *iv)* 10 focam no modelo algébrico ; *v)* 9 mencionam o spark ; *vi)* 8 citam o uso de proveniência ; *vii)* 5 mencionam o caráter declarativo da linguagem de especificação do *workflow* ; *viii)* 3 mencionam tolerância a falhas ; *ix)* 2 tratam explicitamente de UDF ; *x)* 2 mencionam o *SparkSQL* ; *xi)* 1 relaciona o *Apache Flink* como alternativa DISC para execução de *workflow*.

Devido à grande quantidade de artigos criou-se alguns critérios adicionais de filtragem. Analisando o resumo dos 412 documentos foram descartadas as soluções para: *i)* um setor específico da indústria/academia (Ex: *workflow* específico para genética); *ii)* um algoritmo em particular, tal como *decision tree, pagerank, Hash Join*, etc.; *iii)* hardware específico (GPU, FPGA, SoC); *iv)* métodos próximos ao hardware (paralelização a nível de instruções, ou cache do processador); *v) mobile clouds* pois, apesar da relevância atual, são muito específicos; *vi)* problemas de *scheduling* de tarefas em ambiente distribuídos pois o *Spark* já é responsável por isso; *vii)* paralelismo para supercomputadores, pois pressupõe hardware monolítico e especializado; *viii)* custo de uso de recursos em nuvem, pois neste caso o principal objetivo é contábil/financeiro; *ix)* tempo real e *streaming*, apesar do *Spark* apoiar desenvolvimento deste tipo com o modulo *Spark streaming*, pois tira o foco do nosso problema.

O restante foi classificado da seguinte forma: *i)* fonte primária, envolvendo um ou mais dos seguintes temas: *MapReduce*, Álgebra, eDSL, *dataflow*, *Spark* (core, SQL, GraphX, R) independente da área de aplicação *ii)* apoio a proveniência; *iii)* apoio a visualização; *iv)* otimização em banco de dados; *v)* processamento de grafos; *vi)* varredura de parâmetros.

Assim, foram selecionados da lista 33 arquivos para leitura completa, por terem sido publicados após o ano de 2014, portanto, um ano antes da publicação do artigo sobre *SparkSQL* e Catalyst de Armbrust et al. [2015]. Este critério foi adotado pois a intenção é contribuir com algo novo para a Ciência da Computação buscando soluções no estado da arte. No entanto, ao ler alguns desses artigos observou-se a necessidade do estudo de outros artigos mais antigos, tal como de teoria de compiladores e de banco de dados.

Em paralelo a esta busca sistemática foi feita uma pesquisa *ad hoc* visando encontrar às iniciativas específicas relacionados a *workflow* em ambiente *Spark*, que poderiam estar relacionadas a dissertação corrente. Esta pesquisa *ad hoc* revelou que o módulo MLLib do *Spark* apoia o desenvolvimento de análises estatísticas e de aprendizado de máquina e implementa classes para facilitar o desenvolvimento de *workflow* para varredura de parâmetros.

Apresenta-se a seguir uma exposição sobre alguns desses importantes artigos. A classificação das máquinas de execução de *workflows* em ambiente de dados em larga escala, segundo Ogasawara [2011] são: (i) orientadas a varredura de parâmetros, (ii) orientadas a *MapReduce*, (iii) orientadas a coleções aninhadas, (iv) de escalonamento baseado em fluxo, (v) de escalonamento baseado em recursos, (vi) de escalonamento adaptativas, (vii) declarativas e (viii) abordagens algébricas. Pode-se observar uma `ortogonalidade` entre as diversas abordagens classificadas. Isto revela a natureza interdisciplinar e complexa do problema. Faz-se necessário então, optar por domínios específicos do problema e focar o estudo nestes domínios para produzir uma contribuição relevante a área de estudo: Gerência de Dados via Sistema de Gerência de *Workflow* para aplicações de análise intensivas em dados em ambiente paralelo e distribuído. Isto inclui aplicações científicas e comerciais não fazendo distinções entre elas. Porém, restringe-se apenas a *workflows* orientados a dados (*dataflow*) pois apenas neste caso pode-se aplicar os operadores algébricos que regem UDF propostos por Ogasawara [2011].

Dentre os diversos domínios do problema citados anteriormente, foram escolhidos,

para estudo detalhado, os seguintes:

1. **Abordagem algébrica** - a abordagem algébrica com apoio a definição de UDF (como proposto por Ogasawara [2011]) com sua robustez matemática permitindo o uso de regras determinísticas além de heurísticas e de dados de proveniência para otimização do *dataflow*.

   Para esta abordagem também existem outras propostas como o SOFA que é capaz de fazer anotações semânticas em UDF de forma automática ou manual (chamada abordagem híbrida) e permite encontrar conjuntos de planos de execução semanticamente equivalentes para um dado fluxo de dados. As anotações automaticamente detectadas e as criadas manualmente são avaliadas por um otimizador baseado em custos, que usa um conjunto de modelos de reescrita para inferir planos de execução semanticamente equivalentes [Rheinländer et al., 2015].

   As operações algébricas que regem as atividades definidas nas UDF abrem oportunidades para a otimização do fluxo de trabalho permitindo o uso de programas de terceiros ou legados, como implementadores de atividades.

2. **Abordagem *MapReduce*** - a abordagem *MapReduce* permite usar todo o potencial da escalabilidade horizontal do ecossistema *SparkSQL*/*Hadoop* além de todos os tipos de fontes de dados implementados nos métodos de acesso, tais como *Hadoop Distributed File System* (HDFS), arquivos (XML, JSON, Parquet, Avro, Arrow [1], CSV, TSV), WebServices e JDBC (Bancos de dados relacionais e de Séries Temporais). Isto expande a aplicabilidade da solução a virtualmente qualquer aplicação de *bigdata*.

   Alguns *Workflow Management System* (WMS) tradicionais apoiam o desenvolvimento no modelo *MapReduce* para execução paralela e neste caso usam o *Hadoop*. Seguem 3 deles: Kepler [Wang et al., 2009], VisTrails [Callahan et al., 2006] e CloudBLAST [Matsunaga et al., 2008]. Cada um destes WMS tem sua abordagem particular no uso do *Hadoop* para despachar execuções de tarefas Map e Reduce em paralelo nos ambientes distribuídos. Uma vez que as funções *Map* e *Reduce*

[1] O Apache Arrow é um formato para dados em memória com organização colunar, padronizado e independente de linguagem (pode ser usado com C, C++, Java, JavaScript, Python e Ruby) para dados planos (*flat*) e hierárquicos permitindo troca de informação entre processos diferentes. Ele também fornece bibliotecas computacionais e mensagens de fluxo contínuo com cópia zero (*zero-copy streaming messaging*) e comunicação inter-processos.

precisam ser programadas dentro da estrutura do *Hadoop*, o *workflow*, nestas soluções, depende fortemente da implementação *MapReduce* e do código-fonte usado para as atividades, gerando um acoplamento forte entre a definição/especificação do problema e a plataforma de execução do *workflow*, o que é indesejável.

3. **Abordagem eDSL (Declarativas)** - os trabalhos relacionados a linguagens de especificação de *workflow* são bastante relevantes para esta dissertação e podem ser divididas em dois grandes grupos, a saber: i) linguagens gráficas e ii) linguagens textuais.

   As linguagens gráficas usam diagramas e ferramentas interativas para criar os workflows. Devido a sua representação, estas linguagens gráficas apresentam dificuldades na Gestão de Mudança, que é uma área da Engenharia de Software conhecida como *Software Configuration Management* (SCM) e por isso não é abordado neste trabalho.

   As linguagens textuais se valem de sentenças escritas em formato XML, JSON ou texto-plano para informar as dependências de dados e/ou fluxos de controle. As linguagens que detalham o fluxo de controle são usadas em *workflows* centrados no controle (*control-centric*) e as linguagens que detalham as dependências dos dados são usadas em *workflows data-centric*. As linguagens *control-centric* estão fora do escopo deste trabalho, portanto restringe-se aqui o domínio do problema às linguagens textuais para *workflows data-centric*.

   Chen et al. [2015] discorre sobre a diferença entre os possíveis paradigmas de programação que são: *i)* imperativo, onde o usuário especifica exatamente como a computação precisa ser executada e *ii)* declarativo, onde a especificação do usuário se concentra em "o que"deve ser feito. Para ilustrar a diferença e as vantagens das abordagens, considere o exemplo de uma simples expressão de atribuição $a \leftarrow b + 1$ que faz parte de um programa maior. No paradigma imperativo a expressão é compilada e executada atualizando o valor de a de forma imediata. No paradigma declarativo a expressão é compilada em uma representação intermediária, (tal como um grafo), e executada posteriormente, mas apenas quando completar a construção da representação intermediaria de todo o programa, e após todas as otimizações realizadas. A segunda abordagem gera oportunidades adicionais de otimização que podem considerar todo o programa em vez de apenas a estrutura

da expressão de atribuição. Isto é particularmente importante em consultas em *dataflow*, onde otimizações algébricas mudam a seletividade e cardinalidade das relações intermediárias.

Alexandrov et al. [2015] propõem uma eDSL em Scala chamada *Emma* onde os autores implementam *K-means* paralelizado em pouco mais de 20 linhas usando *for-comprehensions*. Esta linguagem forneceu intuições para o projeto (*design*) da eDSL desenvolvida no *framework* WfF.

Em se tratando de usar a infraestrutura de execução do *Apache Spark*, optou-se pela definição de uma DSL interna - eDSL, que pode ser especificada em linguagem Kotlin [2]. A abordagem eDSL facilita o uso por cientistas de áreas distintas (Economia, Biologia, História, Ciências Políticas, Linguística, Administração, Direito, etc.) que em geral não possuem conhecimento técnico para lidar com problemas de infraestrutura de TI e precisam de uma linguagem simples e adequada ao problema da análise de dados.

4. **Abordagem de Varredura de Parâmetros e API de *pipeline* da MLLib** - os algoritmos de aprendizado de máquina geralmente envolvem uma sequência de pré-processamento de dados, extração de características (*feature extraction*), ajuste de modelo (*model fitting*) e as etapas de validação (*validation stages*). A MLLib fornece apoio nativo para o conjunto diversificado de funcionalidades necessárias para a construção de *pipeline* de aprendizado de máquina com dados em larga escala. Esta API usa o ecossistema *Spark* para processar uma *pipeline* de ponta a ponta (*end-to-end*) e com isso simplifica o desenvolvimento do aprendizado em vários estágios, fornecendo um conjunto uniforme de APIs de alto nível [Meng et al., 2016], incluindo APIs que permitem que os usuários troquem por uma abordagem padrão de aprendizado de máquina em vez de usar seus próprios algoritmos especializados. Outros benefícios da API é a otimização de hiperparâmetros (*hyperparameter optimization*)

   O diagrama da Figura 15 mostra os componentes do *Spark MLlib*. A maioria das funções relacionadas à álgebra linear [3] é baseada em uma biblioteca de processa-

[2] A linguagem Kotlin, assim como Scala, também executa na JVM e apoia desenvolvimento de DSL além de ser mais simples e foi escolhida para definição do *workflow* pelo usuário

[3] Breeze - `https://github.com/scalanlp/breeze` ; netlib-java - `https://github.com/fommil/netlib-java` ; F2J - `http://icl.cs.utk.edu/f2j` ; ATLAS - `http://math-atlas.sourceforge.net` ; Intel MKL - `https://software.intel.com/mkl` ; OpenBLAS - `https:`

mento numérico em linguagem Scala (`Breeze`), porém algumas delas são baseadas diretamente na biblioteca de baixo nível `netlib-java` que também é usado pelo `Breeze`. Além disso, o Spark MLlib também possui algumas implementações internas não dependentes do `BLAS`. No `netlib-java`, as implementações do `BLAS/LAPACK` são fornecidas pelo projeto `F2J` - *FortranToJava compiler*. O elemento `system-provided BLAS`, em verde, deve ser instalado separadamente e pode ser um dentre estes : *i)* ATLAS; *ii)* Intel MKL; *iii)* OpenBLAS; ou *iv)* vecLib no caso do macOS.

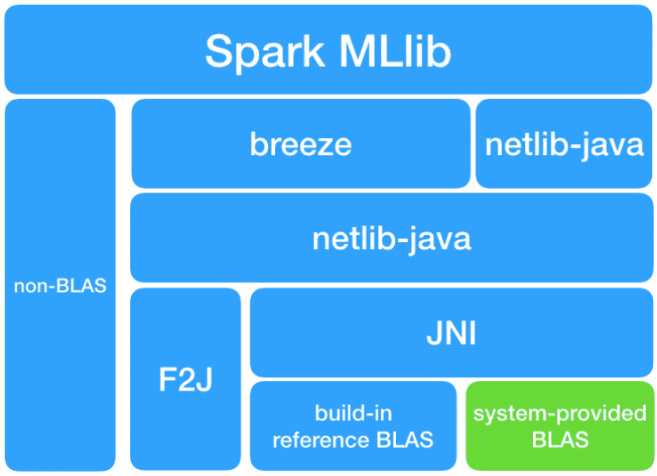


Figura 15: Diagrama apresentando as diversas camadas da implementação MLlib que depende do Breeze que é uma biblioteca Scala que estende a Scala Collection Library para fornecer apoio no desenvolvimento com manipulação de vetores e matrizes. O `Breeze` usa `netlib-java` para acessar as implementações nativas do `BLAS/LAPACK`.

Em termos práticos, a API de *pipeline* da MLLib [4] apoia a seleção de modelos usando ferramentas como *CrossValidator* e *TrainValidationSplit*, que por sua vez usam os itens: *i)* Estimador (*Estimator*) que é o algoritmo (ou pipeline) para sintonizar os parâmetros do modelo; *ii)* um conjunto de *ParamMaps* que são os parâmetros a serem escolhidos ou "grade de parâmetros"; *iii)* Avaliador (*Evaluator*) que fornece a métrica para medir o desempenho de um modelo ajustado considerando os dados de teste.

Recentemente o *Apache Spark* adicionou um novo modelo de agendamento chamado *Barrier Scheduling* [5] o que permite a incorporação de treinamento de *Deep*

`//github.com/xianyi/OpenBLAS` ; vecLib - `https://developer.apple.com/documentation/accelerate/veclib`

[4] O manual de uso pode ser consultado neste link `https://spark.apache.org/docs/2.4.0/ml-tuning.html`

[5] Detalhes sobre *Barrier Scheduling* podem ser vistos em `https://issues.apache.org/jira/`

Learning, distribuído em um estágio do *Spark*. Assim, é possível simplificar um workflow de treinamento de *Deep Learning* distribuído via *Spark MLlib*. Este novo mecanismo de agendamento foi necessário pois o modelo de computação de aprendizado de máquina é diferente do *MapReduce* usado pelo *Spark*. No *Spark*, uma tarefa em um estágio não depende de outras tarefas no mesmo estágio e, portanto, pode ser programada de forma independente. No *Message Passing Interface* (MPI) todos os processos começam ao mesmo tempo e trocam mensagens entre si. Com o estágio de barreira o *Spark* inicia as tarefas ao mesmo tempo e fornece aos usuários informações e ferramentas suficientes para incorporar o treinamento de *Deep Learning* distribuído. Com isso é possível, por exemplo, incorporar um programa MPI usando o Tensor Flow como *back-end*, em um estágio de barreira (*barrier stage*). Além disso as otimizações baseadas nas heurísticas da álgebra de *workflow* poderão usufruir de técnicas de programação usando este novo mecanismo de agendamento de tarefas do *Apache Spark*.

Na avaliação dos trabalhos relacionados considerando o escopo desse projeto, procurou-se dar um direcionamento adequado considerando os quatro domínios descritos anteriormente com a preocupação de avaliar o estado da arte para um direcionamento correto de forma a contribuir na fronteira do conhecimento, ao mesmo tempo que abrindo oportunidades para trabalhos futuros na área.

`browse/SPARK-24374`

(Página intencionalmente deixada em branco)

Capítulo 3 *framework WfF* - Desenvolvimento da Solução

Como foi visto na Introdução, o problema principal a ser considerado neste trabalho é referente a execução otimizada de *dataflows* contendo UDF. Solucionar este problema é a principal contribuição, porém, para facilitar o uso pelos usuários foi criado o *framework* WfF com uma eDSL propria. Neste capítulo descreve-se a solução adotada na implementação deste *framework*, cujos oito requisitos funcionais podem ser divididos em dois grandes grupos:

1. **Otimização algébrica de *dataflow* com UDF no Spark** - cujos quatro requisitos funcionais são: *i)* executar os *dataflows* de análise de dados em ambiente DISC, mais especificamente no *Apache Spark*; *ii)* permitir a especificação de UDF no *workflow*, usando modelo *MapReduce* onde as UDF são regidas por operadores algébricos, compatíveis com a álgebra de *workflow* de Ogasawara et al. [2011]; *iii)* aceitar UDF escrita em linguagem Python, R e código disponível na JVM nas linguagens Java, Scala e Kotlin [Ferreira et al., 2017]; *iv)* permitir, de forma parametrizada, a inclusão de otimização de *dataflow* usando a álgebra de *workflow* juntamente com informações de proveniência retrospectiva colhida do *Apache Spark* via *SparkListener*

2. **Projeto de eDSL e transparência da MDA** - cujos quatro requisitos funcionais são: *i)* prover um modelo independente de computação - CIM, um modelo independente de plataforma - PIM e um modelo específico de plataforma - PSM, como definidos na MDA; *ii)* levantar os requisitos para uma linguagem agnóstica, baseada nos conceitos da MDA, que permita geração de código em diferentes linguagens de programação, particularmente em Scala, R e Python, compatíveis com os módulos *SparkSQL*, *Spark R* e *PySpark* respectivamente; *iii)* definir a linguagem agnóstica como uma eDSL em Kotlin [1] para especificação dos *workflows* pelos usuários alvo: cientistas e engenheiros de dados; e *iv)* executar *workflows* comerciais, industriais e científicos dos tipos ETL, varredura de parâmetros, analise em séries temporais,

[1] Como foi visto na Seção 1.8 a linguagem Kotlin é mais simples e facil de usar quando comparada a linguagem Scala e sua base de usuário é muito maior, apesar de ser uma linguagem mais recente

mineração de dados em *IoT*, dentre outros.

Os requisitos do primeiro grupo estão diretamente relacionados ao tema de pesquisa. Os requisitos do segundo grupo estão relacionados a facilitação do uso pelos cientistas e engenheiros de dados do *framework* WfF.

Todos estes requisitos listados acima foram considerados na análise e definição da arquitetura da solução. Porém, para efeito de implementação o escopo foi limitado considerando o tempo disponível para o desenvolvimento do projeto, que foi dividido em fases. Particularmente a otimização de *dataflow* foi limitado a execução de filtros (operadores wFilter) que operam sobre UDF usando a API *Catalyst* do *Spark SQL* e o casamento de padrões (*pattern matching*) da linguagem Scala. Outras fases serão desenvolvidas em trabalhos futuros, graças ao princípio aberto/fechado (*Open/Closed Principle*) [2] da engenharia de software. Outras limitações estão descritas na Seção . Há que se destacar a aderência do *framework* ao padrão de projeto (*design pattern*) inversão de dependência (*dependency inversion*) típico em *framework* de software.

Com base nos conceitos descritos no Capítulo 1 - *Referenciais Teóricos*, e na avaliação exploratória preliminar foi possível definir uma estratégia para o desenvolvimento do *framework* WfF. Optou-se por criar um contêiner de serviços gerenciados de acordo com uma arquitetura *Service Oriented Architecture* (SOA) para desacoplar os diversos módulos, melhorando a manutenção do software e permitindo um desenvolvimento em fases. Sobre o contêiner de serviços foram construídos os seguintes módulos: *i)* Persistência de *Workflow*; *ii)* Configuração e Ativação de *Workflow* e *iii)* Execução e Monitoramento de *Workflow*. Estes módulos apoiam se em serviços para prover as funcionalidades necessárias dos bastidores (*back-end*). Usa-se um servidor que fornece as funcionalidades via interface *RESTful* baseada no conceito *Representational State Transfer* (REST).

Nas próximas seções apresenta-se o desenvolvimento da solução incluindo uma descrição em alto nível sobe a arquitetura, uma descrição dos módulos que compõem o *framework* e os diagramas UML essenciais ao entendimento da arquitetura proposta. Em seguida, discute-se como é feita a geração de código inspirada na MDA. Um exemplo de uso da eDSL desenvolvida é apresentado para ilustrar qualitativamente a facilidade provida pela linguagem. Apresenta-se também a gramática da eDSL do WfF. Por fim, é apresentado o processo de otimização de operadores com UDF no *Spark SQL*, para o

[2] O princípio aberto/fechado, de Bertrand Meyer especifica que entidades de software (classes, módulos, funções, etc.) devem ser abertas para extensão, mas fechadas para modificação [Meyer, 1988].

caso de operadores *wFilter* contíguos.

3.1 Arquitetura da Solução

A proposta da SOA é baseada em nove princípios, listados a seguir: *1)* Contrato de Serviço Padronizado - Serviços devem aderir a uma descrição não ambígua do serviço; *2)* Baixo Acoplamento - Os serviços devem minimizam dependências uns dos outros; *3* Abstração de Serviço - Serviços devem esconder do mundo exterior, a lógica que eles implementam, ou seja, devem prover encapsulamento; *4)* Reusabilidade de serviço - A lógica deve ser dividida em serviços ortogonais com o objetivo de maximizar a reutilização de código; *5)* Autonomia do Serviço - Os serviços devem ter controle sobre a lógica que eles implementam e encapsulam; *6)* Estado de Serviço - Idealmente, os serviços devem ser sem estado (*stateless*); *7)* Descoberta de Serviço - Os serviços podem ser descobertos (geralmente em um registro de serviço); *8)* Composição de Serviço - Os serviços podem ser compostos por outros, ou seja, é possível quebrar problemas complexos em pequenos problemas mais simples e solucioná-los separadamente; *9)* Interoperabilidade de Serviço - Os serviços devem usar padrões de segurança e de comunicação que permitam o uso por diversos interessados.

Para obter estes resultados é comum implementar um contêiner de serviços para gerenciar o ciclo de vida destes. Além disso um componente, chamado *Service Bus* deve estar disponível para integração dos serviços via modelo assíncrono *publish/subscribe* permitindo programação reativa. Segurança, monitoramento e gerenciamento formam uma outra camada nesta arquitetura. A Figura 16 descreve as camadas dessa arquitetura.

3.2 Desenvolvimento da Solução

No WfF o usuário não interage diretamente com os serviços, uma vez que o programa *Command Line Interface* (CLI) permite a submissão de comandos específicos usando uma eDSL própria para especificação de *Workflow* como será visto na Seção 3.4.

A Figura 17 mostra os componentes de software que compõem o *framework*. Em termos de artefatos binários da solução tem-se dois arquivos JAR, sendo um deles o arquivo *wffcli.jar*, usado pelos clientes para se comunicar com a plataforma. O outro é o servidor que rodará o contêiner de serviços, materializado no artefato *wsc.jar* [3].

[3] A versão 11 da JVM é utilizada por ser a mais recente, mas o código binário é compatível com Java 8

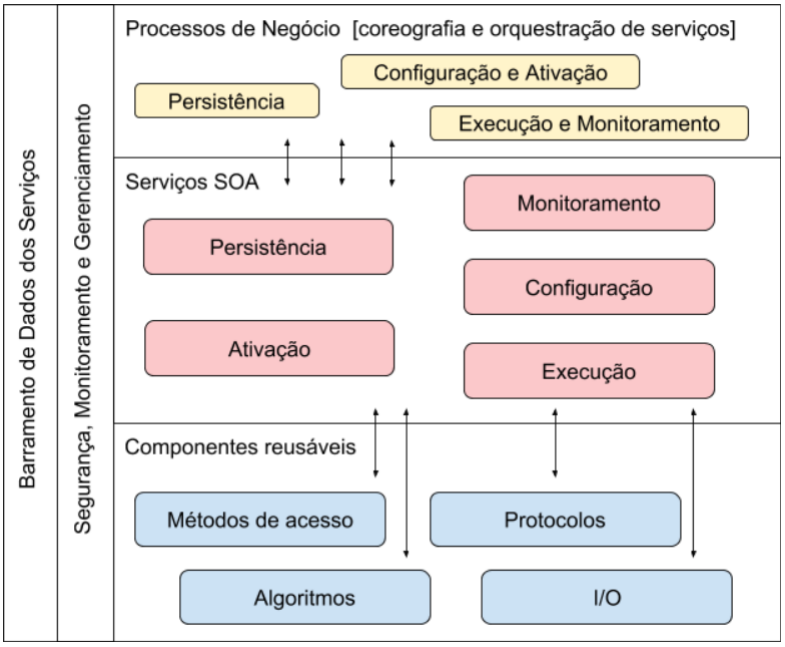


Figura 16: Diagrama de Arquitetura do WfF. Não existe acoplamento entre os processos de negócio e os componentes reusáveis pois a comunicação entre eles é feita pelos serviços.

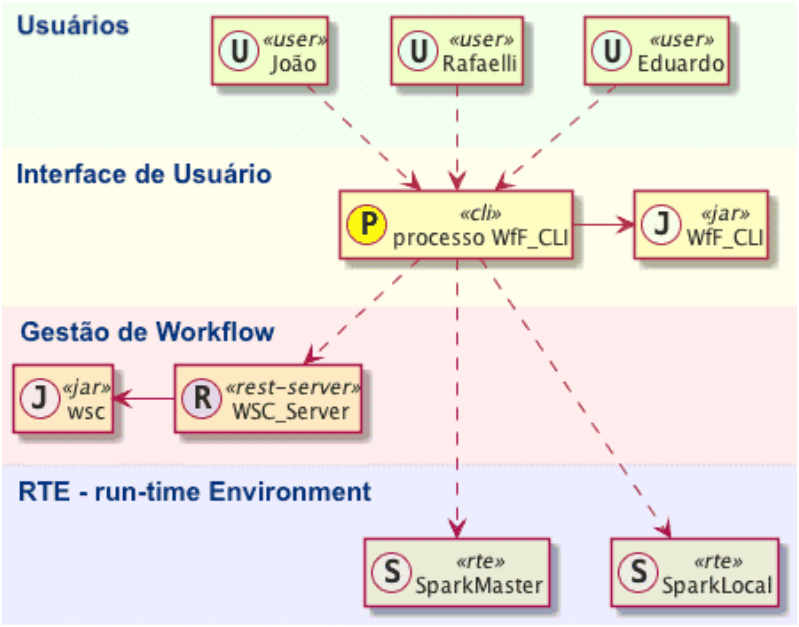


Figura 17: Diagrama de componentes de software e integração com usuários do *framework* WfF.

O componente de Gestão de Workflow é o cerne do WfF. Ele recebe uma mensagem do *WfF_CLI* e distribui via o Barramento de Dados dos Serviços. Assim uma mensagem de início de sessão é processada pelo módulo de configuração e ativação. Em seguida

ou superior

uma mensagem de gravação de fragmento de grafo é tratada pelo modulo de persistência. Ao final uma mensagem de submissão ao *cluster* é tratada pelo modulo de execução e monitoramento que também trata da coleta de proveniência retrospectiva e gravação em repositório próprio [4]. Estes módulos, por outro lado, usam os serviços de forma desacoplada, publicando no barramento de dados. Poderá haver também acessos diretos entre módulos e serviços de mesma natureza de forma síncrona, como por exemplo funcionalidades de persistência entre o modulo e o serviço. O objetivo do barramento de dados é o desacoplamento entre domínios ortogonais de serviços e módulos.

Na Figura 17 ilustra-se como o usuário acessa os serviços do *framework* WfF. Ele instala em seu computador o Java, o compilador Scala 2.11 (ou 2.12) e opcionalmente o *Spark* 2.4.0 (versão de produção mais recente) [5]. Em seguida faz o download do arquivo *wffcli.jar* e dessa forma fica apto a executar o código Kotlin que usa a eDSL já citada. Para isso ele invoca o *wffcli.jar* passando o endereço IP do *host* onde se encontra o WSC_Server (*wsc.jar*), que roda o contêiner de serviços, e implementa as funcionalidades de persistência e geração de código Scala relativo ao *workflow* definido. O usuário pode usar uma *Integrated Development Environment* (IDE) , tal como IntelliJ Idea, para facilitar a edição do código do *Workflow* e ganhar produtividade. O código gerado pelo servidor é devolvido ao usuário na forma de um arquivo compilado *.jar* que pode ser submetido localmente ou à uma instalação remota do *Spark* tal como um *cluster* gerenciado pelo *Standalone Mode*, *Yet Another Resource Negotiator* (YARN) ou *Kubernetes* (k8s). Observe na Figura 17 que o componente responsável pela gestão de *workflow* não se comunica diretamente com ambiente de *run-time* RTE. Em vez disso ele devolve ao usuário o código compilado. Esta é uma simplificação para a fase 1 do projeto. Na fase 2 será possível delegar ao gestor de *workflow* a função de *Spark Driver*, desde que o gestor possa redirecionar *stdin*, *stdout* e *stderr* para o `WfF_CLI`. Este direcionamento deverá ser feito sob *hyper-text transfer protocol* (HTTP) adicionando mais complexidade e por este motivo será implementado na fase 2, pois não está relacionado diretamente com a hipótese de pesquisa.

O *Spark* disponibiliza um *script* chamado *spark-submit* que permite que se submeta uma tarefa ao *cluster* de forma parametrizada. Este *script* usa a classe *SparkSubmit* do pacote *org.apache.spark.deploy*, e este processo pode ser totalmente gerenciado pelo

[4] Na avaliação experimental apresentada no Capítulo 4, os dados de proveniência foram instrumentados, ou seja definidos *hard-coded* , mas é possível coletá-los usando a API *SparkListener*

[5] A única dependência do *framework* WfF é em relação as extensões do *Catalyst* disponíveis a partir da versão 2.2 do *Apache Spark*

framework em versões futuras desde que se use diretamente a classe *SparkSubmit* de dentro do *framework* WfF.

O objetivo de criar uma solução multicamadas para o *(*framework) WfF é prover flexibilidade na implantação da solução, pois o servidor poderá executar *on-promise* ou no modelo *Software as a Service* (SaaS), em infraestruturas acessíveis publicamente tal como Azure, GoogleCloud, AWS ou Digital Ocean.

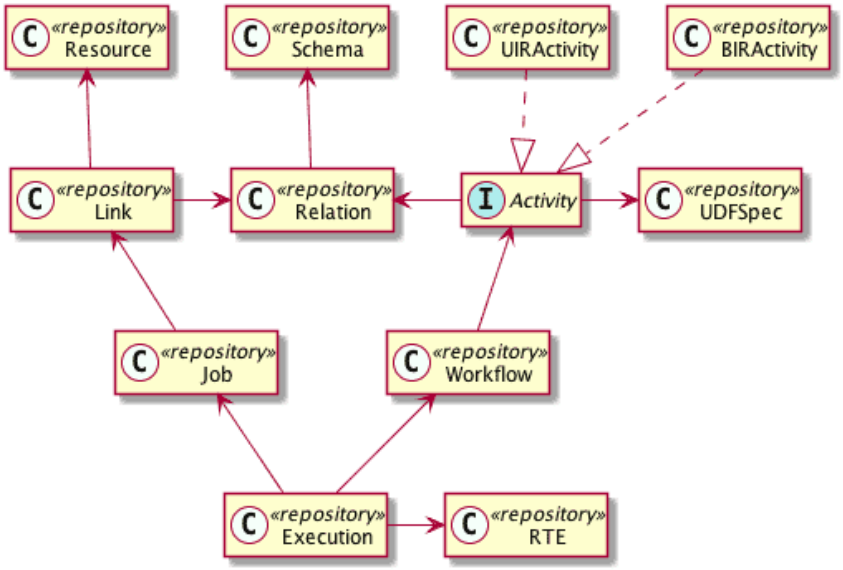


Figura 18: Diagrama UML apresentando as dependências entre as principais classes e interfaces usadas no serviço de persistência do *framework*. A classe `RTE` representa o *Run-time Environment* e possui várias instâncias de objeto onde se armazena todas as configurações conhecidas de ambiente de *run-time* do *Apache Spark*. As outras classes refletem fielmente o que foi especificado no *dataflow* escrito usando a eDSL definida pelo *framework* WfF.

No *framework* WfF um certo *dataflow* é especificado pelo usuário através de um texto usando a eDSL. Este texto é considerado um programa WfF. Este programa é traduzido para uma representação de grafo pelo serviço de persistência. A Figura 18 mostra um diagrama de classes no padrão UML que ilustra as dependências entre as diversas classes no domínio do serviço de persistência. Assim, as instruções contidas no programa WfF se transformam em registros no repositório do *framework*, organizados em tabelas, como definidas no diagrama. Um exemplo de programa WfF aparece nas Figuras 24, 25, 26 e 27 da Seção 3.4.

As classes no diagrama refletem os comandos definidos na eDSL. Uma observação adicional deve ser feita em relação a interface `Activity` que foi necessária para abstrair dois tipos de atividade disponíveis no *framework*. Essas atividades são implementadas

nas classes: *i)* `UIRActivity` que aceita apenas uma relação de entrada e serve para registrar atividades relacionadas aos operadores *wMap*, *wFilter*, *wSplitMap*, *wGroupBy* e *wReduceByKey* e *ii)* `BIRActivity` que aceita duas relações de entrada (esquerda e direita) e serve para registrar atividades relacionadas ao operador *wJoin*.

A funcionalidade de persistência de *workflows* no repositório usa também um outro conjunto de classes relacionadas ao grafo de dependências do *dataflow*. As classes responsáveis por tratar este problema aparecem no diagrama UML da Figura 19. As responsabilidades de cada uma das classes e o relacionamento entre elas são como segue:

`Relation` - Armazena as informações sobre os nomes das relações e seus esquemas.

`Activity` - Interface de abstração para atividade com apenas uma relação de entrada *Unary Input Relation Activity* (UIRA) e atividade com duas relações de entrada *Binary Input Relation Activity* (BIRA).

`UIRActivity` - UIRA armazena dados de dependência no *dataflow*, do operador que rege a atividade e da UDF usada pelo operador algébrico.

`BIRActivity` - BIRA armazena dados de dependência no *dataflow*, do operador que rege a atividade, que é sempre wJoin, e do campo/atributo de junção.

`GraphNode` - Classe abstrata que generaliza os conceitos de atividade e relação que são os itens que aparecem no Grafo. Esta classe mantém o atributo *vertexId* que é um identificador do objeto `Vertex` associado ao `GraphNode` em foco. Esta classe é responsável por manter a ligação entre o objeto `Vertex` e este `GraphNode` correspondente. O `GraphNode` funciona como um classificador na persistência para prover reusabilidade de código.

`Vertex` - Esta classe representa um elemento vértice do grafo. Tal elemento possui como atributos: *i) name* - o mesmo nome do `GraphNode` associado; *ii) type* - o tipo que identifica o papel daquele `GraphNode` no grafo de dependências. Este tipo pode ser *InputRelation*, *InputAndOutputRelation*, *OnlyOutputRelation* [6], `Activity`, etc; *iii) primaryType* - o tipo primário que só recebe uma das duas opções (`Relation` ou `Activity`) e serve apenas para facilitar a implementa-

[6] *OnlyOutputRelation* ocorre uma única vez no grafo

ção da navegação no grafo e por último; *iV) dbId* é uma referência para o ID do `GraphNode` no banco de dados.

Ao inserir o elemento `Vertex` no repositório deve-se atualizar o `GraphNode` correspondente com o *id* do `Vertex` inserido para manter essas referências cruzadas íntegras e consistentes.

Em termos de algoritmos a maior complexidade diz respeito a representação do grafo e sua navegação (rotinas de caminhamento *Breadth-First Walk*). Apesar do *Spark* prover a API *Graphframes* para manipulação de grafos (descrito na Seção 1.1), optou-se por representá-lo por dois mapas de adjacências, sendo um para navegar partindo das relações de entrada e chegando até a relação de saída, passando por todas as atividades, e o outro na direção oposta.

Foram criados algoritmos recursivos com filas de apoio que registram os nós já visitados. O grafo que representa um *dataflow* é um típico DAG com algumas restrições que ajudam a simplificar a implementação. Por exemplo, não importa quantas relações de entrada existam, tem-se sempre uma única relação de saída.

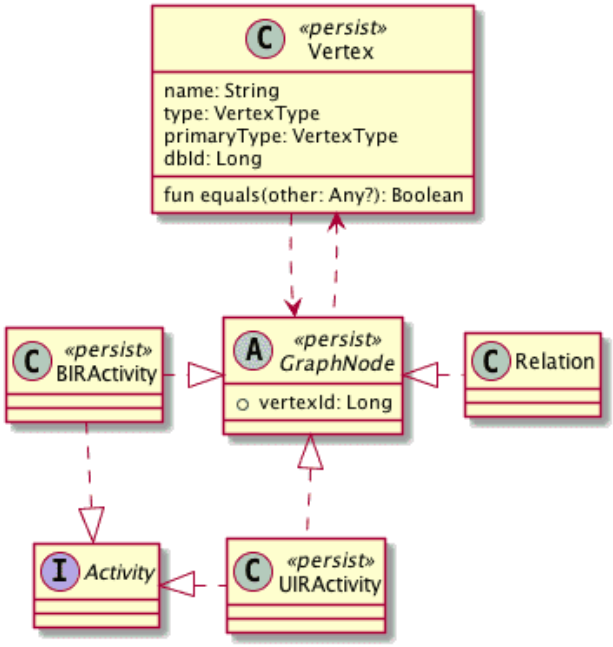


Figura 19: Diagrama de classe (UML) apresentando as dependências entre as principais classes e interfaces usadas no serviço de persistência do *framework* relacionadas ao domínio do grafo de dependências do *dataflow* associado.

3.3 Geração de código inspirado na arquitetura MDA

Com o grafo pronto e as rotinas de caminhamento *Breadth-First Walk* testadas, pode-se passar à discussão sobre a geração de código. Tudo o que aparece aqui, considerando a geração de código para linguagem Scala, pode ser replicado para linguagem Python e para linguagem R, considerando as particularidades gramaticais de cada linguagem, como se vê a seguir.

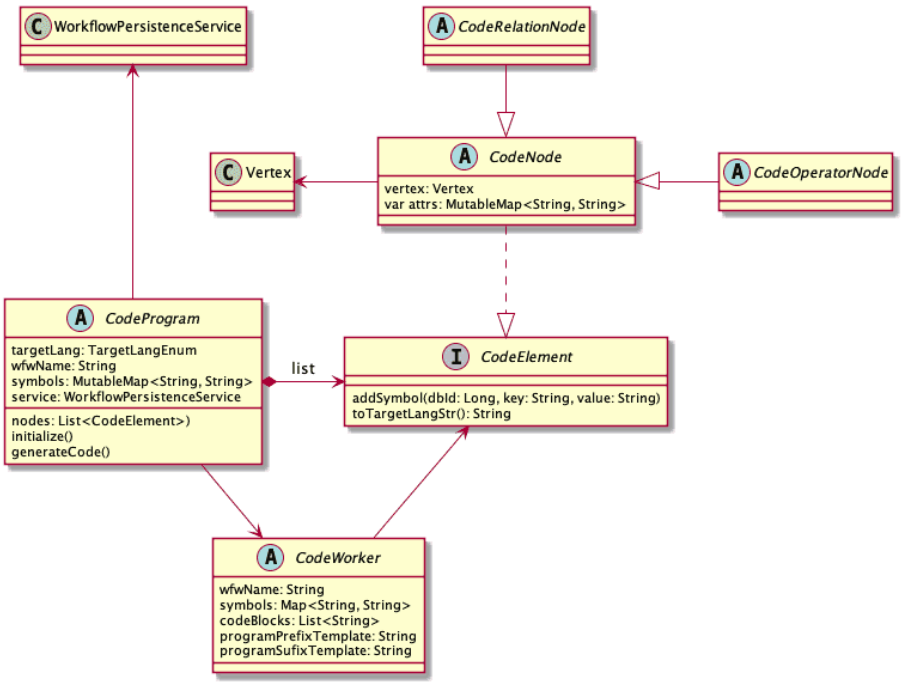


Figura 20: Diagrama de classes UML relacionando classes concretas, classes abstratas e interfaces que colaboram para gerar o código compatível com o *Apache Spark*.

A geração de código do *framework* é feita graças a colaboração das classes descritas no diagrama UML da Figura 20, onde cada classe tem sua responsabilidade como segue: *i)* `CodeProgram` é a classe abstrata que coordena o processo de geração; *ii)* `CodeWorker` é a classe abstrata que colabora com a classe `CodeProgram` na tarefa de construção do programa, armazenando e mesclando modelos (*templates*) de código; *iii)* `CodeNode` é uma classe abstrata e suas especializações são responsáveis por tratar a especificidade do código de acordo com o tipo de vértice relacionado, ou seja, para um vértice do tipo *InputRelation* é gerado um código a partir de um *template* específico, ao passo que para uma atividade relacionada a um operador (tal como *wMap*) é gerado um código a partir de outro *template*, e assim por diante.

Além disso, as classes abstratas `CodeProgram`, `CodeWorker`, `CodeNode`, `CodeRelationNode`

e `CodeOperatorNode` possuem especializações da linguagem alvo que como foi dito, podem ser Scala, R ou Python.

A especialização `ScalaCodeWorker` que estende `CodeWorker` define, dentre outras coisas, *templates* para algumas funções úteis, pois neste caso, alguns *implicts* do *SQLContext* entram em conflito com outros do objeto *Expression* (ambas classes do *Spark*). Veja na Figura 21 dois exemplos de função de apoio, onde a primeira função constrói um *AttributeReference* a partir de um *Symbol* relacionado a um valor *Integer* e a segunda faz o mesmo para um *AttributeReference* relacionado a um valor *String*.

```
import org.apache.spark.sql.types.IntegerType,StringType,DoubleTye e,LongType
import org.apache.spark.sql.catalyst.expressions.Literal,Attribute Reference

def int(s: Symbol): AttributeReference =
   AttributeReference(s.name, IntegerType, nullable = true)()

def str(s: Symbol): AttributeReference =
   AttributeReference(s.name, StringType, nullable = true)()
```

Figura 21: Exemplo de código Scala de apoio criado pelo *framework* para funções de apoio que é necessário para evitar conflitos de *implicits* da linguagem Scala.

O armazenamento do *template* para a geração de código da classe de otimização de *dataflow* também é de responsabilidade da classe `CodeWorker` e suas especializações. Atualmente a otimização depende de código Scala do *Catalyst*, e portanto, reside na classe `ScalaCodeWorker`.

Como foi visto, o programa é gerado com o apoio de modelos (*templates*) armazenados como cadeia de caracteres multi-linha em código Kotlin da classe `CodeWorker` e suas especializações. No caso da linguagem Scala a estrutura do programa gerado pelo WfF é organizada em blocos da seguinte forma: *i)* declaração e importação de pacotes gerais usando comando *package* e *import*; *ii)* declaração e definição das classes de otimização usadas, pois são agnósticas ao problema; *iii)* prefixo com a declaração do objeto *Workflow* usando o padrão de nome da seguinte forma: `Workflow_WorkflowName` para evitar conflitos entre *workflows* diferentes existentes na persistência; *iv)* prefixo com a declaração do método *main* que é padrão da JVM para Java, Scala e Kotlin e funciona como ponto de entrada do programa *entrypoint* a ser submetido ao *Spark*; *v)* código relacionado a criação da *SparkSession* [7]; *vi)* código dinâmico do *dataflow* traduzido da

[7] No caso do usuário especificar na eDSL que deseja otimização, a *SparkSession* será criada com esta funcionalidade

eDSL ; *vii)* código para desconectar-se da sessão corrente no *Spark* para evitar vazamento de memória; *viii)* classes utilitárias que colaborem com o *workflow* na execução do *dataflow* usando delegação que ajuda na reusabilidade de código.

No escopo deste projeto foi testado apenas a geração de código em linguagem Scala. Como foi visto, a estrutura é flexível e provê reusabilidade de código e a geração de código para a linguagem R e/ou Python poderá ser desenvolvida em trabalhos futuros.

3.4 Exemplo de uso da eDSL desenvolvida

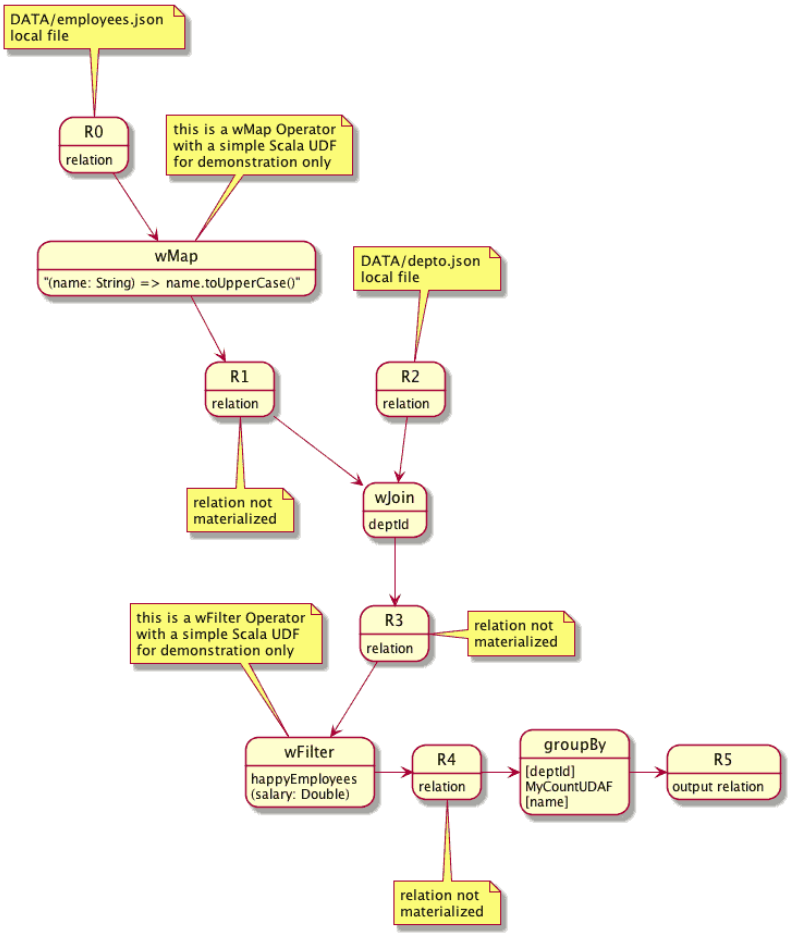


Figura 22: *Dataflow* de exemplo ilustrando a definição de um *workflow* de análise de dados.

Para facilitar o entendimento da solução desenvolvida considere o exemplo mostrado na Figura 22. Este *dataflow* serve para ilustrar a funcionalidade dos serviço de definição, monitoramento e execução de *workflows*. O *dataflow* mencionado tem como requisito duas relações de entrada e três funções de usuário. As relações de entrada $R0$ e $R2$ estão materializadas em arquivos JSON, mas poderiam vir de qualquer recurso definido

```
df.createOrReplaceTempView("R4")
val R5 = spark.sql(
    "SELECT MyCountUDAF(name) as myCountUDAF FROM R4 GROUP BY
'deptID'")
R5.show()
```

Figura 23: Código Scala para o cliente usando a UDAF em uma consulta *(*query).

por uma URI (ou seja podem ser tabelas HDFS, *Java Database Connectivity* (JDBC), *Parquet*, etc.). As funções de usuário servem aos operadores wMap, wGroupBy e wFilter e foram escritas em Scala para facilitar o teste, porém é possível escrevê-las em Python, Kotlin, Java, Go ou R, graças a utilização de envelopes (*wrappers*) como descrito na seção 3.3. No Capítulo 4, foi implementada uma UDF em Kotlin para realizar o experimento e validar a hipótese de otimização de *dataflow* pelo WfF.

No *dataflow* mostrado na Figura 22, na etapa de agregação de dados, usou-se a função de agregação *MyCountUDAF* construída sobre uma UDAF, e para tal criou-se, em linguagem Scala, uma *TempView* e usou-se o método *sql* do objeto *DataFrame*, como mostrado na Figura 23. A técnica para criação da UDAF foi descrita na Seção 1.3. A UDAF é especificada e implementada pelo usuário e fornecida como recurso para o *framework* [8].

A agregação de dados no *Apache Spark* é bem flexível e abrangente. O *Spark* permite agregar dados de tuplas em valores escalares e tipos complexos como *arrays* e listas. Também pode atuar em toda a população ou em amostragens. Além de média, mínimo, máximo e desvio padrão as funções *skewness*, *kurtosis*, *covariance* e *correlation* também são implementadas. Com este apoio ao desenvolvimento de UDAF para o WfF pode-se escrever UDF de agregação em outras linguagens de programação usando um código Scala como envelope (*wrapper*), ampliando ainda mais as funcionalidades de agregação permitidas no *dataflow*.

Este exemplo mostrado na Figura 22 pode ser definido pelo usuário usando a eDSL do *framework* WfF. Para isso, define-se o *workflow* de forma abstrata, sem especificar os recursos materializados das relações $R0$ e $R2$. Também não é necessário informar o código externo que implementa as UDF, bastando informar as assinaturas dessas UDF. A Figura 24 apresenta essa definição. Um *workflow* na sua forma abstrata define um nome

[8] Mais detalhes do exemplo com UDAF pode ser visto no repositório GitHub em `https://github.com/joao-parana/wff-exp-aval`

e um conjunto de itens que podem ser quaisquer entre os tipos: relação (`relation`), atividade (`activity`) e uma especificação de assinatura de UDF (`udfSpec`). A ordem em que esses itens aparecem na lista `workflowItems` não é relevante, pois o que define a dependência entre estes elementos são as Atividades. Sempre que um dado *workflow* é inserido no repositório via serviço de persistência, o grafo é construído/atualizado onde são identificadas todas as relações de dependência.

```
Workflow name "W1" with
       workflowItems {
         relation name "R0" schema "S0 = deptId: Long, name: String, salary: Double"
         relation name "R1" schema "S1"
         relation name "R2" schema "S2 = deptId: Long, deptName: String"
         relation name "R3" schema "S3"
         activity name "A1" wMap using("R0", "R1", "udf0")
         activity name "A2" wJoin using("R1", "R2", "R3", "deptId")
         activity name "A3" wFilter using("R3", "R4", "udf3")
         activity name "A4" wGroupBy using("R4", "R5", "deptId", "MyCountUDAF", "name")
         // See belows UDFs signatures specifications:
         udfSpec name "udf0" signature "nameSmartCased(name: String): String"
         udfSpec name "udf3" signature "happyEmployees(salary: Double): Boolean"
         udfSpec name "MyCountUDAF" signature "myCountUDAF(fieldName: String): Long"
         result willBe "R5" // Only for documentation and checking.
       }
```

Figura 24: Fragmento de programa WfF que especifica o *workflow* de análise de dados de forma abstrata.

Observe que na especificação das relações de entrada é necessário definir o esquema com a lista de campos e tipos de dados correspondentes, e que nas relações intermediárias, fruto da execução das atividades, basta informar um nome desejado para o esquema.

Este *workflow* abstrato não pode ser executado sem as especificações dos recursos a serem usados. Estes recursos podem ser arquivos locais, tabelas de bancos de dados, arquivos disponíveis em *clusters Hadoop* ou arquivos de código-fonte das UDF, etc. Veja os detalhes na Figura 25 para o caso de uma execução do nosso *dataflow* de teste mostrado na Figura 22

Este conjunto de comandos `resource` define onde cada um dos recursos usados no *dataflow* podem ser encontrados. No caso das UDFs é necessário primeiro definir onde pode ser encontrado o programa que implementa as funções e depois pode-se fazer referência a estes programas e ao nome de cada UDF. Neste exemplo, hiperlinks são usados apontando para um *serviço do GitHub para hospedagem de trechos de código* (GIST), pois fornece o benefício adicional de gestão de mudança, e assim, é possível gerenciar a versão exata que está sendo executada numa dada instância de `Job` (conceito que será apresentado a seguir). Observe que na URL da `SCALA_SOURCE_CODE` aparece

```
Resources {
  resource name "res0" title "tabela empregados" type JSON_FILE uri "file://DATA/employees.json"
  resource name "res2" title "tabela departamentos" type JSON_FILE uri "file://DATA/depto.json"
  // supondo um arquivo fonte com todas as UDFs dentro do mesmo arquivo fonte Scala
  resource name "sourceCode_1" title "Scala source code for UDF" type
     SCALA_SOURCE_CODE uri "https://gist.github.com/joao-parana/f39f1a4f7480eb3b4ec1a896bea32d3c"
  resource name "sourceCode_2" title "Scala source code for UDAF" type
     SCALA_SOURCE_CODE uri "https://gist.github.com/joao-parana/30fa32d916720d1f10577ac53fa097d8"
  resource name "resUdf0" udf "smartTextCase" type SCALA_FUNCTION uri "sourceCode_1"
  resource name "resUdf3" udf "happyEmployees" type SCALA_FUNCTION uri "sourceCode_1"
  resource name "resUdf4" udf "MyCountUDAF" type SCALA_CLASS uri "sourceCode_2"
}
```

Figura 25: Fragmento de programa WfF que especifica os recursos a serem usados em uma dada execução do *workflow* de análise de dados definido previamente. Observe que existe um tipo de recurso para um fonte Scala e outro para uma função Scala que reside no fonte definido anteriormente.

um *hash* de 16 bytes em hexadecimal. Este valor identifica univocamente a versão do recurso no repositório de arquivos fontes do GitHub, o que facilita a reprodutibilidade dos experimentos *in-silico*.

Com o *workflow* na sua forma abstrata e a localização dos recursos associados pode-se criar instâncias de `Job` que poderão ser submetidos para execução em um dado ambiente de execução *Run-time Environment* (RTE). Isto é mostrado na Figura 26 onde o `Job` especifica a ligação entre as relações e UDFs, e os seus recursos materializados. Cada comando `link` possui um tipo que pode ser `data` ou `code` e as ligações que se deseja entre as informações abstratas de relações, UDF e seus correspondentes recursos, previamente definidos.

A separação da definição do *dataflow* nestas três categorias permite reusabilidade de código, combinações arbitrárias de instâncias de execução e reprodutibilidade de experimento em diversas situações possíveis, ampliando a capacidade de análise com o uso do *framework* proposto. Outra vantagem é a capacidade de dividir as responsabilidades pela manutenção do ambiente entre o engenheiro e o cientista de dados. Os cientistas de dados podem se responsabilizar pela categoria *`Workflow`*, enquanto o engenheiro de dados manipula as categorias `Resource`, `Job` e `Execution`. A separação de responsabilidades pode facilitar a colaboração entre as equipes.

Por último, a Figura 27 mostra como é possível especificar as execuções propriamente ditas definindo o RTE alvo. Esta é uma característica chave da MDA (como descrito na Seção 1.7) e implementada no *framework* WfF.

No exemplo descrito o mesmo *workflow* será submetido a três ambientes diferentes usando o mesmo `Job` [9]. A execução em ambiente local é boa para análise explora-

[9]Outra opção seria criar três `Jobs` diferentes e a gramatica da eDSL aceita essa construção, provendo

```
Jobs {
 job name "Job1" title "usando JSON no Spark." config {
   link name "rlk0" data using( "R0"  , "res0")
   link name "rlk2" data using( "R2"  , "res2")
   link name "udflk0" code using("udf0", "resUdf0")
   link name "udflk3" code using("udf3", "resUdf3")
   link name "udflk4" code using("MyCountUDAF", "resUdf4")
 }
}
```

Figura 26: Fragmento de programa WfF que especifica os *Jobs* a serem executados em um experimento com *workflow* de análise de dados definido previamente.

```
val experimentName = "Experimento in-silico 'funcionários felizes'."
val experimentTitle = """
 Este experimento in-silico visa testar um workflow de análise de
 dados em três ambientes diferentes coletando métricas de desempenho
 """.trimIndent()
Execution {  // Rodando o Job no ambiente Local com 3 núcleos (cores)
 app name experimentName title experimentTitle
 run job "Job1"  collecting metrics
 workflow name "W1"
 local cores 3
}
Execution {  // Rodando o Job no Cluster Kubernetes com 36 núcleos (cores)
 app name experimentName title experimentTitle
 run job "Job1"  collecting metrics
 workflow name "W1"
 cluster kubernetes "220.12.43.79" pod "spark36" cores 36
}
Execution {  // Rodando o Job no Cluster Spark padrão com 32 núcleos (cores)
 app name experimentName title experimentTitle
 run job "Job1"  collecting metrics
 workflow name "W1"
 master ip "210.2.33.9" cores 32
}
```

Figura 27: Fragmento de programa WfF que especifica os execuções de um dado experimento usando um *workflow* de análise de dados definido previamente.

tória inicial da avaliação da metodologia. Os `Jobs` executando no *cluster* podem usar *datasets* muito maiores e podem usar dados de produção.

É importante salientar que nesta abordagem todas as funcionalidades da linguagem hospedeira podem ser usadas dentro do código da eDSL, código este, que é mais "limpo"e focado na descrição do *dataflow*. Como foi visto, o código é dividido em seções (`Workflow`, `Resource`, `Job` e `Execution`) permitindo tratar a abstração da definição do `Workflow`, independente de onde ele será executado e quais recursos materializados serão usados. Assim, é possível, para um mesmo `Workflow`, executá-lo em ambientes diferentes com *datasets* diferentes, desde que represente o mesmo domínio, é claro. Isso permite também reprodutibilidade de experimentos *in-silico*.

A Figura 12, da Seção 1.7 mostra as modificações realizadas na arquitetura MDA original necessárias à realidade específica do *framework* WfF. Na modificação proposta

flexibilidade.

a intervenção humana ocorre na especificação do *dataflow* pelo cientista de dado (no CIM), ou seja o código mostrado na Figura 24. Este CIM no formato texto plano da eDSL é compilado e convertido para um DAG que é armazenado no repositório usando o Serviço de Persistência do *framework*. Este DAG é a representação PIM do *dataflow*, e é construído usando o CIM e as Diretrizes Específicas de Domínio parametrizadas no *framework*. Este PIM, aliado a especificação da ligação (A) entre o PIM e o PSM, feita pelo cientista de dado via eDSL, e também aos padrões e regras específicas de plataforma, fornecem ao *framework* as bases para geração do PSM. A ligação (B) entre o PSM e o código gerado é fornecida pelo cientista de dado (ou engenheiro de dado) via eDSL, e o *framework* consulta seu repositório para encontrar os padrões específicos de linguagem (Scala na versão atual e Python/R nas versões futuras), para gerar corretamente o programa a ser enviado ao *Spark* no RTE. Vale ressaltar que toda a interação humana no *framework* é feita através de linguagem específica de domínio com comandos de semântica específica e restrita, em um alto nível de abstração, usando os elementos `Workflow`, `Resource`, `Jobs`, `Execution` e suas composições. Essa característica pode ser vista nos fragmentos de programa das figuras 24, 25, 26 e 27. Ressalta-se ainda que o cientista de dados pode contar com um engenheiro de dados para ajudar com as especificações dos elementos `Resource`, `Jobs` e `Execution` da eDSL quando estes exigirem informações específicas (e desconhecidas) de plataforma de execução. Assim, a MDA como implementada no *framework* favorece a separação de responsabilidades (*Separation of Concerns*) entre os cientistas e os engenheiros de dados no contexto DISC para análise de dados.

3.5 A gramática da eDSL do WfF

É útil que se tenha uma definição formal da gramática para orientar os usuários na forma de especificar os *workflows*. Para tal foi escolhida a metasintaxe EBNF descrita na Seção 1.8. A Figura 28 traz a definição formal para a gramática observa-se a pequena quantidade de *tokens* utilizada, assim como a expressividade semântica de cada *token*. Esta riqueza semântica é requisito em projeto de linguagens específicas de domínio.

No desenvolvimento de linguagens de programação costuma-se projetar regras semânticas que ajudam no processo de geração de informação ao usuário (*feedback*) sobre problemas no programa definido. Isto costuma ser feito para melhorar o tratamento de erro, facilitando a correção do programa, ou seja, apenas regras gramaticais não são

```
Workflow ⇐ "Workflow", "name", Identifier, "with", Identifier, NewLine,
           WorkflowItems, NewLine;
WorkflowItems ⇐ "workflowItems", Identifier, "{", WorkflowItemList, "}";
WorkflowItemList ⇐ ( Relation | Activity | UDFSpec | Result ), NewLine;
Relation ⇐ "relation", "name", Identifier, [ Schema ];
Schema ⇐ "schema", SchemaDef
       | "schema", Identifier
       ;
SchemaDef is a String defining the schema
Activity ⇐ "name", Identifier, ( "wMap" | "wFilter" | "wJoin" | "wGroupBy"|
           "wSort" ) , "using(", Params , ")";
Params ⇐ Identifier, Identifier, Identifier
       | Identifier, Identifier, Identifier, Identifier
       | Identifier, Identifier, Identifier,  Identifier, Identifier, Identifier
       ;
UDFSpec ⇐ "udfSpec", "name", Identifier, "signature" UDFSignature ;
UDFSignature is a String defining the schema
```

Figura 28: Definição da gramática (na metasintaxe EBNF) para o elemento *workflow* da eDSL do *framework* WfF.

suficientes para obter uma eDSL robusta, no entanto, este tema não será tratado neste trabalho e poderá ser desenvolvido em trabalhos futuros.

3.6 Otimização de operadores com UDF no *Spark SQL*

Como foi visto anteriormente na Seção 1.1, o Spark usa a API *Catalyst* para prover otimização de consultas em ambiente DISC e disponibiliza pontos de extensão no objeto *SparkSession*. Os pontos de extensão devem ser implementados como classe na linguagem Scala e deve usar casamento de padrão para realizar as otimizações, com o cuidado de não alterar a semântica da consulta. Viu-se também que as regras da álgebra constituem um método robusto para especificação de heurísticas de otimização. Além disso, na Seção 1.4 foi mostrado que em teoria de Banco de Dados, especificamente na Álgebra Relacional, usa-se várias regras algébricas para efeito de otimização de consultas. Portanto, faz-se necessário formular tais regras para o caso de consultas com UDF regidas por operador *wFilter*, que é o escopo deste trabalho. Assim, inicia-se esta seção descrevendo as regras que se aplicam as UDF regidas por operador wFilter. Vale destacar que a lista de regras abaixo não esgota o assunto e que apenas as regras

associadas diretamente a otimização de filtros contíguos com UDF, tratada na avaliação experimental, foram apresentadas aqui. Outras regras serão objetos de trabalhos futuros.

Regras gerais de transformação da Álgebra de *Workflow*

1. Cascata de $wFilter$ - Um predicado conjuntivo de $wFilter$ pode ser desmembrado em uma sequência de operações $wFilter$ individuais e vice-versa. (Regra 1).

$$wFilter(c1 \bigwedge c2, R) \equiv wFilter(c2, wFilter(c1, R)) \quad (1)$$

Assume-se aqui que a execução da UDF $c1$ no contexto da atividade regida pelo operador wFilter, não gere nenhum efeito colateral que possa alterar o resultado da atividade $c2$ e vice-versa. Isto é possível pois as relações no *Spark* são objetos imutáveis (*immutable*) e portanto, esta limitação é bastante razoável para UDF escalares e de agregação.

2. Comutatividade de Seleção - As operações de $wFilter$ são comutativas. Observe que a Regra 2 estabelece uma equivalência semântica, porém o custo de execução das duas configurações da consulta (esquerda ou direita) podem ser muito distintos.

$$wFilter(c1, (wFilter(c2, R)) \equiv wFilter(c2, (wFilter(c1, R)) \quad (2)$$

3. Cascata de Projeção - Não existe equivalência nem analogia para esta regra na Álgebra de *Workflow* exceto pelo fato de que uma relação R_f obtida de $wFilter(c1, R)$ obedece a Regra 3) abaixo.

$$\pi_{c1}(\pi_{c2}(\pi_{c3}(R_f))) \equiv \pi_{c1}(R_f)) \quad (3)$$

A Regra 3 é util para que o WfF faça projeções, de forma transparente, a partir das especificações dos esquemas das relações (`relation`) de entrada e correspondentes especificações dos recursos (`resource`), a fim de simplificar o esquema das relações de saída.

As três regras listadas acima são suficientes para a implementação da otimização proposta no *framework* WfF

Com as regras algébricas em mãos é possível definir a classe a ser usada pelo *Catalyst*. Como já foi mencionado, o *Apache Spark* foi adotado para apoiar os usuários na execução de *workflow* de análise de dados com UDF, em ambientes de processamento distribuído em larga escala [Ferreira et al., 2017]. Uma das vantagens desta abordagem é deixar a responsabilidade de toda a complexidade do modelo de execução, incluindo resiliência e desempenho, para o *Spark*. Isso diferencia a proposta de Ferreira et al. [2017] dos outros SGW que assumem esta responsabilidade de gestão do ambiente de execução e distribuição de tarefas entre os nodes do *cluster*. Desta forma, o foco da integração com o *Spark*, no contexto da otimização, restringe-se a anotação semântica de UDF, para prover informação necessária a otimização do *workflow* em tempo de execução. Para isso, foi realizada uma análise exploratória das APIs do *Catalyst* e dos pontos de extensão (citados na Seção 1.1) para otimizações com UDF regidas por operadores wFilter.

Para compreender o experimento *in-silico* realizado neste trabalho, imagine um problema que possui um fragmento de *workflow* com duas UDF regidas por um operador $wFilter$, como mostrado na Instrução 4 em linguagem Scala.

A otimização já implementada no *Spark* (chamada *CombineFilters*) permite transformar a instrução 4 na outra instrução 5, pois trata-se de dois filtros adjacentes, conforme a Regra 1, que foi apresentada na Seção 1.5 - álgebra de *workflow*.

$$dataset.filter(slowerUDF(some, parameters)).filter(fasterUDF(param1, ...)) \quad (4)$$

$$dataset.filter(slowerUDF(some, parameters)\&\&fasterUDF(param1, ...)) \quad (5)$$

onde $\&\&$ é o operador lógico AND

Esta transformação não altera a semântica da relação e nem a relação produto-consumo dos dados. É importante dizer que na fase de geração de código o gerador

de *byte-code* do *Spark SQL* otimiza o *dataflow* usando a regra 1, juntamente com conceitos de álgebra booleana, gerando um desvio (*branch*) de curto circuito entre as duas expressões. [10]

Apesar de todas as otimizações implementadas pelo *Spark* ainda resta um problema: este código avalia primeiro uma função dispendiosa ($slowerUDF$) [11] e, em caso desta retornar $true$, segue para avaliar o restante da expressão. Quando o primeiro teste na $slowerUDF$ falha, a avaliação sofre o curto-circuito, mas ainda assim existe uma penalidade de desempenho significativa, pois o custo computacional de $slowerUDF$ pode ser muito grande e afeta o processamento de todas as tuplas da relação (*dataset*) de entrada. Neste ponto, entra em cena a Regra 2 que estabelece as bases para a otimização heurística proposta, que é a de execução da UDF dispendiosa, sobre um conjunto menor de tuplas, ou seja, é necessário usar a propriedade de comutatividade da seleção.

Com o objetivo de otimizar este tipo de *workflow* foi criada uma classe (chamada *ChangePositionOfTwoAdjacentsFiltersContainingUDF*[12]) escrita em linguagem Scala para funcionar como ponto de extensão. Ela consiste de uma *case class* do Scala[13] e é responsável por transformar planos lógicos por meio de otimizações algébricas. O objetivo desta classe é trocar a posição de dois operadores *Filter* adjacentes. Isso só é possível se os operadores atuam sobre UDF anotadas com semântica de custo de execução diferenciado, e se existe ganho de desempenho relevante no *workflow* com a troca. Para isso, foram implementados dois métodos: *hasTwoUDFOneInEachFilter* e *apply*. O primeiro método avalia dois *Filters* e devolve um valor booleano, indicando se os dois atendem os requisitos em relação a regerem ou não UDF anotadas. O segundo é o ponto de entrada do *Catalyst* na classe de otimização personalizada. Este método *apply* recebe um plano lógico e deve retornar outro plano lógico de acordo com os requisitos de interface do *Catalyst*.

A Figura 29 mostra o casamento de padrão para o caso da otimização de *filter* contendo UDF. Ressalta-se que é necessário fazer o casamento de padrão ao mesmo tempo em que os valores dos objetos são extraídos e atribuídos como um todo em uma variável. Este casamento de padrão é conhecido como ligação de variáveis em cláusulas

[10] Esta otimização com desvio de curto-circuito é usada também em otimização de código em compiladores

[11] Soares and Souza [2017] descrevem exemplos de treinamento de redes neurais que podem ser considerados como UDF dispendiosas

[12] O código fonte Scala da implementação pode ser acessado no repositório público no endereço https://github.com/joao-parana/wff-catalyst

[13] a classe *ChangePositionOfTwoAdjacentsFiltersContainingUDF* estende a classe *Rule[LogicalPlan]*

case (*Binding Variables in case Clauses*). Na Figura 29, as variáveis *f*, *ab* e *ff* indicam, respectivamente, o primeiro *Filter*, o *AnalysisBarrier* e o segundo *Filter*. Elas são acessadas no código que processa o casamento do respectivo padrão para que se consiga alterar a árvore de regras do plano lógico de acordo com a necessidade.

```
case f @ Filter(condition: Expression, ab @ AnalysisBarrier(ff @ Filter(_, grandGChild)))
     if hasTwoUDFOneInEachFilter(f, ff) => {
```

Figura 29: Casamento de padrão para dois operadores *Filter* com um elemento *AnalysisBarrier* entre eles. O método *hasTwoUDFOneInEachFilter* verifica se os filtros estão aptos a otimização.

É importante salientar que o método *hasTwoUDFOneInEachFilter* implementa um predicado que verifica o nome da UDF contra tabela de uso de recursos, mantido pelo subsistema de proveniência retrospectiva, para decidir pela troca de posição dos filtros. A troca só será feita se for vantajosa, segundo os critérios estabelecidos. Os dados de proveniência de interesse para a otimização referem-se apenas ao nome da UDF e o custo computacional médio de execução medido em milissegundos. Na fase 1 do projeto este subsistema foi instrumentado *hard-coded*.

O *Catalyst* funciona no modelo/padrão inversão de controle, assim, esta regra criada participa no processo de otimização da mesma forma que qualquer outra regra do *Catalyst*, ou seja, no caso em que o *dataflow* satisfaz a condição para otimização algébrica citada, o *Spark* promove a troca e o *dataflow* continua sendo otimizado por outras regras, nativas ou não.

Ao criar uma regra no *Catalyst*, deve-se tomar o cuidado de não gerar repetição infinita (*loop*) na avaliação da regra, já que o casamento de padrão implementado poderá ser invocado diversas vezes e é tipicamente *stateless*.

Neste capítulo foram abordados diversos aspectos relacionados ao desenvolvimento da solução incluindo o objetivo principal do tema de pesquisa (otimização de *dataflow* e a API *Catalyst*), mas também foram descritas as técnicas usadas para definição da eDSL e sua implementação em Kotlin. Abordou-se também a MDA e a geração de código Scala.

O próximo Capítulo apresenta os resultados experimentais obtidos com os programas gerados pelo *framework* WfF, como descritos neste capitulo.

(Página intencionalmente deixada em branco)

Capítulo 4 Avaliação Experimental

A avaliação experimental realizada procurou avaliar o *framework* proposto quanto a otimização de *dataflow* usando álgebra de *workflows* e uma classe em Scala que é invocada pela API *Catalyst* do *Apache Spark*, em um paradigma de inversão de controle, como descrito na Seção 3.6 - (Otimização de operadores com UDF no *Spark SQL*).

Foi escolhido um *dataset* sintético provido pelo *benchmark* TPC-H pois ele fornece ferramentas de geração de *datasets* para *bigdata* com cardinalidades vaiadas e de forma parametrizada, o que permite gerar *datasets* com 15 mil clientes ou com mais de 150 milhões de clientes. Dessa forma, o TPC-H facilita tanto o desenvolvimento de análises exploratórias iniciais em um computador local, quanto a execução de *benchmarks* em ambiente de *cluster*, acessando *datasets* extremamente grandes.

Os experimentos tratam da execução de uma consulta (*query*) do *benchmark* TPC-H com o objetivo de avaliar a otimização algébrica do WfF. Foi escolhida a consulta identificada como `q13`. Essa consulta busca relacionamentos entre clientes e o tamanho de seus pedidos. Ela determina a distribuição de clientes pelo número de pedidos que fizeram, incluindo clientes que não têm registro de pedidos, passados ou presentes. Ela conta e informa quantos clientes não têm pedidos, quantos têm 1, 2, 3 etc. Uma verificação é feita para garantir que os pedidos contados não se enquadrem em uma das várias categorias especiais de pedidos. Categorias especiais são identificadas na coluna de comentários do pedido, procurando por um padrão específico (*tokens* tais como: *special*, *pending*, *unusual*, *express* em conjunto com *packages*, *requests*, *accounts*, *deposits*). O plano de execução dessa consulta foi mapeado como *workflow* que usa vários operadores da álgebra de *workflow* e foi implementado como *dataflow* no WfF. Para comparar a versão não otimizada (padrão do *Apache Spark*) com a versão otimizada, usando a implementação de otimização de filtros contíguos contendo UDF descritas no capítulo IV, foram feitas algumas modificações na semântica da *query* original como será descrito mais adiante na Seção 4.6. Na versão modificada duas UDF implementam o predicado original via um operador *wFilter* e essas UDF recebem, cada uma, um *token* a ser verifi-

cado de sua ocorrência nos comentários dos pedidos.

Neste capítulo, usam-se alguns termos originais em inglês, que são: núcleo de processador (core), computador (host) , node e nó de trabalho (worker) e estes termos podem aparecer também no plural: núcleos de processador (cores), computadores (hosts), nodes e nós de trabalho (workers). Foi escolhida esta abordagem pelo fato de serem amplamente usados na literatura e pela semântica não ambígua.

Na próxima seção apresentam-se os argumentos para a escolha do formato de armazenamento a ser usado nos experimentos, já que o formato original dos *datasets* é *Comma Separated Values* (CSV), que possui baixa eficiência no sistema de arquivos. Nas seções seguintes discorre-se sobre a avaliação experimental tratando das métricas de avaliação, da infraestrutura computacional utilizada e dos *datasets* escolhidos. Na execução dos experimentos detalha-se aquele que avalia a otimização algébrica onde são descritos os *dataflows* `W0`, `W1A` e `W1B`. Ao final aborda-se a diferença de abordagens e a estratégia de otimização do *Spark* e como isso está relacionado ao *framework* WfF, para terminar com uma discussão geral sobre os resultados da avaliação experimental. No anexo B aborda-se os resultados da prova de conceito realizada para avaliação da transparência de *datasource* em relação às características do sistema operacional e das materializações dos *datasets* referenciados no *workflow*, ou seja, está relacionado à transparência provida pela MDA em relação a fonte de dado (*datasource*) na qual o *framework* foi inspirado.

4.1 Escolha do formato de armazenamento

Nesta avaliação experimental, as execuções dos *dataflows* devem ser repetidas dezenas de vezes sobre os mesmos *datasets* alterando o número de cores e medindo o tempo total decorrido, portanto, é necessário escolher um formato de armazenamento adequado para as relações de entrada dos *dataflows*. Para gerar os *datasets* sintéticos é necessário usar o gerador de dados *dbgen* do *benchmark* TPC-H obtido do site http://tpc.org[1]. Tais dados foram importados para o ambiente *Apache Spark* e convertidos para o formato Apache *Parquet*.

A escolha deste formato permitiu obter melhor desempenho no acesso ao disco rígido e melhor eficiência no espaço ocupado, já que as tabelas originais são muito grandes

[1]Foi criado um repositório com os detalhes de como reproduzir o processo em um computador um sistema operacional macOS, Linux ou Windows - https://github.com/joao-parana/TPC-H-2.17.3

(cardinalidade na ordem de 150 milhões de clientes e 537 milhões de pedidos associados) e o formato Parquet é colunar, enquanto o CSV é por linha.

A principal diferença entre formatos orientados a linha (ou registros) e os orientados a colunas é que os orientados a registros (arquivos de texto, formatos delimitados como CSV, TSV) favorecem aplicações transacionais enquanto os orientados a colunas favorecem análise de dados, principalmente em ambiente de *bigdata*. O Parquet trata um problema comum neste ambiente de maneira muito eficiente. É comum ter tabelas (conjuntos de dados) com muito mais colunas do que você esperaria em um banco de dados relacional bem projetado (normalizado). Isso ocorre porque geralmente usamos modelos desnormalizados dos dados para facilitar as consultas, já que algumas junções podem ser resolvidas previamente com os respectivos resultados materializados em *dataset*. Existem outras vantagens, como no caso de armazenamento de dados de estado no tempo (séries espaço-temporais) ou de dados para modelos de aprendizado de máquina com centenas de características (*features*). De qualquer forma, é comum ter um número grande de colunas em uma tabela em sistemas analíticos. Embora a consulta dessas tabelas seja fácil usando o SQL, é comum que se deseje obter um conjunto de registros com base em apenas alguns valores de algumas dessas colunas, e o processo de filtragem é feito por predicados envolvendo uma ou mais colunas.

Esta filtragem em um formato de linha exige a leitura de todo registro, separação das colunas e, por último, a verificação se atende ou não aos predicados. No formato colunar, lê-se apenas o bloco no sistema de armazenamento relacionado a coluna desejada e, portanto, é muito mais eficiente. O Parquet consegue realizar a filtragem na camada de baixo nível que acessa o sistema de arquivos, e o otimizador de *query* do *Spark* (Catalyst) gera o código, transferindo a filtragem de predicados para esta camada de software e com isso é exigido muito menos esforço para leitura do disco rígido.

Com os dados no formato Parquet é possível executar o *dataflow* usando o *framework* WfF e comparar execuções usando as tabelas no formato CSV [2] (geradas pelo TPC-H) *versus* tabelas no formato Parquet (carregadas previamente usando um *dataflow* de ETL). Durante a avaliação experimental, observou-se que os *datasets* no formato Parquet apresentaram taxas de compressão de dados entre 50% e 67% (em relação ao original no formato CSV), dependendo do conteúdo. Isto viabilizou o teste com até 150 milhões de clientes e os correspondentes 537 milhões de pedidos.

[2] Os testes comparativos de acesso a arquivos CSV e Parquet não foram realizados

Como será visto na Seção 4.6, foram realizadas algumas mudanças na *query* original para permitir comparação entre consultas usando *dataflow* com e sem otimização algébrica com código *Catalyst* de otimização de filtros contíguos desenvolvidos para este propósito, como descrito na Seção 3.6.

4.2 Métricas de avaliação

O uso de métricas para avaliação de ambientes de processamento paralelo tem sido assunto de estudo por parte de diversos cientistas da computação e vários indicadores tem sido propostos para medir o desempenho e a escalabilidade no *cluster* [Sun and Gustafson, 1991], [Kumar et al., 1994] e [Hwang and Xu, 1998]. Nesta avaliação experimental, foram usados as métricas: tempo decorrido (*elapsed-time*), *speed-up* e eficiência. A métrica mais importante é a de tempo decorrido, pois serve para calcular todas as outras. Por exemplo um sistema computacional `S1` com oito cores, (1 host com 8 cores, ou 2 hosts com 4 cores cada, ou 4 hosts com 2 cores cada). Nesta configuração de *hardware*, pode-se executar o processo usando apenas um core, e depois 2, 3, 4, 5, 6, 7 e, por último 8. Tem-se, ao final, o conjunto de medidas de tempo decorrido: $T_1, T_2, T_3, T_4, T_5, T_6, ...T_p$ onde p é o número de cores, neste exemplo limitado a 8. A medida de tempo T_1 é chamada tempo de execução serial (*serial execution time*) pois é obtida sem nenhum paralelismo.

A Expressão 6 descreve o conjunto de medidas de tempo tomadas para a execução de um dado processo, quando se escala de 1 até p cores.

$$\textbf{elaspsed\_time} \rightarrow \quad \{T_1, T_2, T_3, T_4, T_5, T_6, ..., T_p\} \tag{6}$$

Em seguida, pode-se calcular o *speed-up*, que é definido pela razão entre o tempo serial pelo tempo paralelo, e permite medir o quanto efetivamente estão sendo usado os recursos dos processadores para resolver um dado problema de modo paralelizado. A Expressão 7 define o *speed-up*. Como já foi mencionado, T_1 é o tempo serial, enquanto que $T_2, T_3, T_4, T_5, T_6, ..., T_p$ são os tempos referentes a execução paralela.

$$\textbf{speed\_up} \rightarrow \quad Sup = \left\{ \frac{T_1}{T_1}, \frac{T_1}{T_2}, \frac{T_1}{T_3}, \frac{T_1}{T_4}, \frac{T_1}{T_5}, ..., \frac{T_1}{T_p} \right\} \tag{7}$$

Por último, pode-se calcular a eficiência. A eficiência ξ é definida como a razão entre o *speed-up* e a quantidade de processadores, como mostra a Expressão 8 que define a

eficiência para a *p-ésima* configuração de cores. Se a eficiência está perto de 1 significa que cada um dos *p* processadores estão sendo usados eficientemente.

$$\textbf{efficiency} \rightarrow \quad \xi_p = \frac{Sup_p}{p} \tag{8}$$

ou seja,

$$\xi_p = \frac{T_1}{pT_p} \tag{9}$$

É importante observar que estas métricas citadas acima são muito simples e não levam em consideração características de rede e de sistema de entrada e saída dos computadores do *cluster*. Além disso, em qualquer aplicação, parte do código pode ser executado em paralelo e outra de forma serial impactando a escalabilidade e o desempenho. Estas métricas simples não levam em consideração essas variáveis. No entanto, no escopo desse trabalho considera-se uma boa aproximação.

4.3 Infraestrutura computacional

Os testes foram executados em três ambientes distintos enumerados a seguir:

1. Análise exploratória no computador local com 4 cores rodando o sistema operacional macOS Mojave e *Spark* versão 2.4.0 com interface REPL do Ammonite

2. Execução de aplicativos Scala auto-contidos via *shell script* spark-submit no computador local com 4 cores rodando o sistema operacional macOS Mojave e *Spark* versão 2.4.0

3. Execução de Aplicativos Scala auto-contidos via *shell script* spark-submit no *cluster* do CEFET-RJ com 24 cores e 34.5 GB de memória RAM rodando sistema operacional Ubuntu 16.04 e *Spark* versão 2.3.0. Os 24 cores são providos por 6 computadores com CPU AMD PRO A10-8750B R7, cada um com 4 cores. O driver *Spark* executa em computador com 4 cores rodando o sistema operacional Ubuntu 16.04 com CPU Intel Xeon E3-1220 V2 @ 3.10 GHz, 16 GB de RAM e *Spark* versão 2.3.0. O armazenamento persistente conta com um HDD ST2000DM006-2DM1 com 1,8 TB em sdb compartilhado via *Network File System* (NFS) entre todos os workers e além disso cada worker conta com um ST1000DM003-1SB1 de 931 GB em sda para armazenamento local.

4.4 Distribuição de tarefas no *cluster*

Para a execução dos experimentos, fez-se a tomada de medidas com as três métricas descritas na Seção 4.2: tempo decorrido (elapsed-time), *speed-up* e eficiência. Como mencionado anteriormente, a medida mais importante é a de tempo decorrido (elapsed-time) pois serve para calcular todas as outras.

A distribuição do processamento entre os nodes é outra informação que pode ser relevante para algumas análises. Veja na tabela mostrada na Figura 30, como os cores são alocados pelo *Spark* a medida que se solicita mais workers (*Executors*). Nesta tabela, a primeira coluna mostra o número de cores utilizados na execução da tarefa. As colunas $\alpha, \beta, \gamma, \delta, \epsilon$ e ζ referem-se aos computadores alpha, beta, gamma, delta, epsilon e zeta do *cluster* e as linhas verticais nas células são as quantidades de cores utilizados em cada computador. O *Spark* pode distribuir o processamento no *cluster* usando um dentre os seguintes: YARN, Mesos, Kubernetes ou *Spark Standalone Scheduler*. Nesta avaliação experimental, foi usado o *Spark Standalone Scheduler* pela sua simplicidade. Observe que o *Spark Standalone Scheduler* usa uma heurística para distribuir os executores de forma distribuir a carga de processamento de forma homogênea. Quando se executa no *cluster* usando um node ele usa um dos cores do processador do host delta, se é pedido mais um ele aloca no host gama usando um core em cada um desses dois hosts. Esse processo continua até que todos os cores estejam sendo usados quando é solicitado que se use os 24 cores.

4.5 *Dataset* escolhido

O *benchmark* TPC-H serve para testes de eficiência em sistemas de apoio à decisão (*decision support benchmark*). Consiste em um conjunto de consultas *ad-hoc* orientadas para negócios com modificações simultâneas nos dados. As consultas e os dados que populam o banco de dados foram escolhidos por ter ampla relevância no setor comercial. Seu objetivo é avaliar sistemas de apoio à decisão que lidam com grandes volumes de dados, executando consultas com alto grau de complexidade respondendo as questões críticas de negócios. Este *benchmark* define sua própria métrica de desempenho (que é a métrica de desempenho de consulta por hora composta, `QphH@Size`) que reflete vários aspectos do consumo de recursos do sistema para processar consultas. No entanto, o objetivo não é comparar o desempenho de consultas executadas no *Spark* com

#	α	β	γ	δ	ε	ζ
1				⊥		
2			⊥	⊥		
4		⊥	⊥	⊥		⊥
6	⊥	⊥	⊥	⊥	⊥	⊥
8	⊥	⊥	⫫	⫫	⊥	⊥
10	⊥	⫫	⫫	⫫	⊥	⫫
12	⫫	⫫	⫫	⫫	⫫	⫫
14	⫫	⫫	⫴	⫴	⫫	⫫
16	⫫	⫴	⫴	⫴	⫫	⫴
18	⫴	⫴	⫴	⫴	⫴	⫴
20	⫴	⫴	⫴⫴	⫴⫴	⫴	⫴
22	⫴	⫴⫴	⫴⫴	⫴⫴	⫴	⫴⫴
24	⫴⫴	⫴⫴	⫴⫴	⫴⫴	⫴⫴	⫴⫴

Figura 30: Distribuição das *tasks* nos nodes do *cluster* a medida que são solicitados mais cores.

as mesmas consultas executadas em outros SGBD, portanto, esta métrica original do TPC-H não foi usada.

Apresenta-se a seguir, uma descrição mais detalhada do catálogo relacionado ao *dataset* escolhido, que descreve a estrutura e o esquema das tabelas e seus relacionamentos.

Para a avaliação da otimização de *dataflows* foram escolhidas duas tabelas do esquema original, usadas na *query* `q13` [3] : Cliente (*customer*) e Pedido (*order*). O diagrama da Figura 31 mostra o relacionamento entre o cliente e seus pedidos. Os itens dos pedidos não estão representados. Estas duas tabelas serão as entradas para os *dataflows* `W1A` e `W1B`, como será visto nas Seções 4.6, 4.6.1 e 4.6.2.

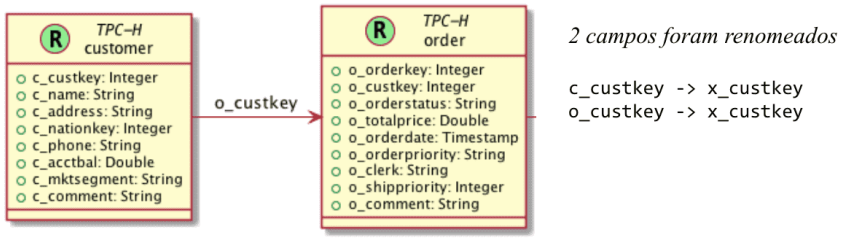


Figura 31: Esquema das tabelas *customer* e *order* do *benchmark* TPC-H: dois campos foram renomeados por conveniência.

É importante destacar que dados gerados através de uma ferramenta (tal como o

[3] Esta consulta foi escolhida por usar uma junção e dois filtros com respectivos predicados que podem ser adaptados para uso em UDF, permitindo avaliar a otimização algébrica

benchmark TPC-H) têm um viés, já que o gerador cria exemplos segundo uma distribuição de probabilidade que pode não corresponder aos "dados reais". Assim, algum cuidado deve ser tomado quando se compara resultados de programas diferentes com dados gerados por simuladores diferentes, pois é provável que cada simulador gere exemplos com diferentes distribuições [Wainer et al., 2007]. Porém não é este o caso desta avaliação experimental pois usamos a *query* `q13` do benchmark TPC-H isoladamente.

O *dataset* de dados sintéticos do *benchmark* TPC-H usado nesta avaliação experimental pode ser criado usando os *scripts* que estão disponíveis no repositório `joao-parana/TPC-H-2.1` [4] do GitHub, em termos de código fonte e binário compilado para o Ubuntu 64 bits. No arquivo README do repositório, descreve-se o esquema do banco de dados envolvendo todas as tabelas, a cardinalidade relativa com um fator de escala (SF - *Scale Factor*) de cada tabela, e os relacionamentos um-para-muitos entre as tabelas.

Na próxima seção, apresentam-se os resultados da execução dos experimentos com as medidas tomadas segundo o descritivo da Seção 4.2. Trata-se dos resultados experimentais para validar as hipóteses propostas nesta dissertação: capacidade de ganho substancial com otimização usando a álgebra de *workflow*

4.6 Experimentos para avaliação da otimização algébrica

Para executar os experimentos de validação da otimização algébrica foi criado um *dataflow*, como este que aparece na Figura 32, em um formato texto plano simples.

```
R1
↓
A1 → R3 → A2 → R4 → A3 → R5 → A4 → R6 → A5 → R7 → A6 →
R8
↑
R2
```

Figura 32: Dataflow para a *query* `q13` do TPC-H.

Este *dataflow* implementa o equivalente a *query* SQL (*query* `q13` do TPC-H) mostrada na Listagem 4.1, já com as devidas modificações [5].

[4] `https://github.com/joao-parana/TPC-H-2.17.3`

[5] Esta *query* mostrada na Listagem 4.1 pode ser escrita diretamente em SQL no *Spark* para avaliar o plano de execução e comparar com a abordagem baseada em *dataset* usada no WfF, que é fortemente tipada.

No *dataflow* da Figura 32 tem-se as seguintes atividades e relações, e seus respectivos papéis:

- a atividade **A1** é regida por um operador *wJoin* do tipo *left-outer* sobre *x_custkey*,
- a atividade **A2** é regida por um operador *wFilter* e usa a UDF *remoteDoNotContains*,
- a atividade **A3** é regida por um operador *wFilter* e usa a UDF *doNotContains*,
- a atividade **A4** é regida por um operador *wGroupBy* e usa a UDF *MyCountUDF*,
- a atividade **A5** é regida por um operador *wGroupBy* e usa a UDF *MyCountUDF* sobre a coluna c_count previamente agregada ,
- a atividade **A6** é regida por um operador *wSort* com as regras custdist/desc, c_count/desc ,
- as relações de entrada são **R1** para *customers* e **R2** para *orders*
- a relação de saida é **R8**,
- as relações **R3**, **R4**, **R5**, **R6** e **R7** são relações intermediárias

Observe na Listagem 4.1 que o predicado original `o_comment not like '%:1%:2%'` pôde ser implementado como UDF. Nesta abordagem, o operador `not like` foi substituído por um operação lógica conjuntiva envolvendo duas UDF regidas por *wFilter* que são: `remoteDoNotContains( o_comment, '%:1')` e `doNotContains(o_comment, '%:2')`. Essas UDF fazem o trabalho de *parser* na *string* `o_comment` para filtrar os registros adequadamente. O predicado em questão será fornecido como parâmetro para o *wFilter* das atividades **A2** e **A3** para comportar o teste de otimização em UDF usando *Catalyst* com a classe [6] projetada para este propósito. Tem-se, então, o segmento de *dataflow* **R3** → **A2** → **R4** → **A3** → **R5** e as respectivas UDF: UDF A - `remoteDoNotContains( o_comment, '1%')` e UDF B - `doNotContains(o_comment, '2%')`

A otimização traz resultado positivo, pois a UDF A está hospedada em um servidor remoto e é acessada via HTTP, e portanto, tem custo de execução consideravelmente

[6] A classe Scala `ChangePositionOfTwoAdjacentsFiltersContainingUDF`, que faz a otimização via *Catalyst*

Listagem 4.1 Código SQL para a *query* `q13` do TPC-H

```
select c_count, count(*) as custdist
from (
  select
    c.x_custkey, count(o_orderkey), o_comment
  from
    customer c left outer join orders o
    on c.x_custkey = o.x_custkey
    -- and o_comment not like '%:1%:2%' --
  group by
    c.x_custkey
) as c_orders (c.x_custkey, c_count, o_comment)
where
  remoteDoNotContains(o_comment, ':1%') and
  doNotContains(o_comment, ':2%')
group by c_count
order by custdist desc, c_count desc;
```

maior que a UDF B, que roda diretamente na JVM. O operador *wSort* foi criado para atender a demanda desta consulta na atividade **A6**.

Há que se mencionar o fato das definições dos esquema de **R1** e **R2** forçarem operações de projeção (da álgebra relacional) para apoiar o desacoplamento entre a definição do *dataflow* e os correspondentes recursos materializados a serem usados. Isto é importante, pois a lista de campos usados na sequência do *dataflow* pode não corresponder em quantidade e posicionamento em relação ao *dataset* equivalente definido como recurso em uma dada instância de execução.

Os experimentos foram realizados com dois grupos de *datasets* sendo um de cardinalidade muito grande e o outro de cardinalidade pequena. Veja na Tabela 4 as características desses *datasets* e a lista de *dataflows* que são executados sobre eles. A tabela apresenta o nome do *dataflow* e os formatos de dados envolvidos.

Tabela 4: Experimentos com clientes e pedidos - cardinalidade dos *datasets*

grupo	ϕ**: clientes**	θ**: pedidos**	**formatos**	***dataflows***
large	150 milhões	537 milhões	csv, parquet	`W0`
short	150 mil	1.5 milhões	parquet	`W0`, `W1A`, `W1B`

Como foi mencionado na Seção 4.1 o formato colunar é mais adequado ao processamento de consultas e por este motivo antes de iniciar os experimentos para a avaliação de otimização algébrica foi executado um *dataflow* do tipo ETL para conversão de formato de armazenamento. A seguir são listados os três *dataflows* que foram executados.

- `W0` é um ETL que converte os dois *datasets* escolhidos do *benchmark* TPC-H - Cliente (*customer*) e Pedido (*order*) para o formato Parquet. Além de avaliar a eficiência do formato Parquet no *cluster* para volumes maiores, este ETL `W0` também deixa as duas tabelas disponíveis para os *dataflows* seguintes, em um formato mais eficiente facilitando futuras *queries*, tal como a *query* `q13`, principal objetivo desta avaliação experimental. Desta forma, este *dataflow* lê o *dataset* customer.csv, gera o *dataset* customer.parquet e em seguida lê o *dataset* order.csv gerando o *dataset* order.parquet. Como já foi mencionado anteriormente, as UDF filtram categorias especiais de pedidos que são identificadas na coluna de comentários do pedido, procurando por um padrão específico (*tokens* tais como: *special*, *pending*, *unusual*, *express* em conjunto com *packages*, *requests*, *accounts*, *deposits*) e por este motivo, este *workflow* adiciona uma nova coluna ao *dataset* order.parquet chamada `special_category` que é um mapa de bits materializado como tipo *Short*. Esta coluna indica quais *tokens* existem no registro e como a coluna foi particionada, foi possível testar o efeito do particionamento na otimização [7].

- `W1A` executa todas as operações com UDF relacionadas a *query* `q13` do TPC-H, e as duas atividades regidas pelos operadores *wFilter* contíguos que possuem UDF implementadas em Scala. O objetivo é avaliar como as UDF impactam o processamento em *cluster*, quando estas atuam sobre um número aproximadamente 10% menor de registros em relação ao total de registros, devido a sequência de filtros sendo executados. Ou seja, o intuito é avaliar o desempenho do *framework* para *workflows* quando o uso dos recursos se concentram em processamento, além de entrada e saída.

- `W1B` é o mesmo *dataflow* que `W1A`, porém usando a otimização de filtros contíguos. O objetivo é avaliar o quanto a otimização usando álgebra de *workflow* traz benefícios aos usuários em termos de desempenho computacional em ambiente distribuído.

[7] O particionamento não apresentou melhorias, possivelmente, devido ao uso do *Spark Standalone Scheduler* e não foram realizadas medições com YARN e Kubernetes para validar a hipótese de melhoria com o particionamento. Isto pode ser objeto de trabalhos futuros.

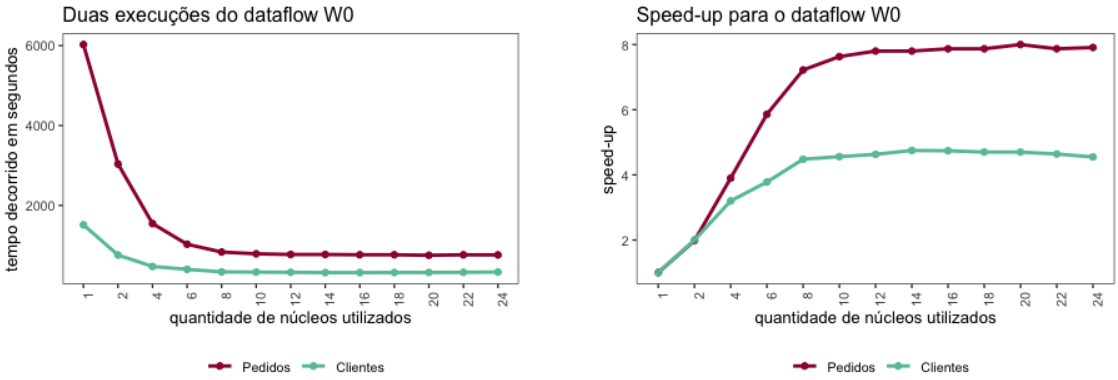
A execução de *dataflow* usando atividades regidas por operadores que invocam UDF via *wrapper* do *ExternalDriver* para código em linguagem R, foi realizada apenas com propósitos qualitativos. Observou-se que a interface TCP/IP do *ExternalDriver* via REngine da implementação JRI adiciona um custo extra, em termos de latência, na faixa de 20 milissegundos por chamada do método *evaluateRCode()*, ou seja, para cada tupla da relação de entrada da atividade. É importante observar que apesar do operador *wFilter* restringir tuplas na saída, todo o *dataset* de entrada sofre essa sobrecarga.

A seguir, descrevem-se os resultados das execuções dos diversos experimentos variando o número de cores e mantendo o mesmo *dataflow* e os mesmos *datasets*. Realiza-se também uma discussão preliminar sobre os resultados.

4.6.1 Preparação para análise da otimização de UDF com Catalyst (`W0`)

Para converter os *datasets customer* e *order* do formato CSV para Parquet, de forma a serem usados nos *dataflows* `W1A` e `W1B` foi criado o *dataflow* `W0` como foi citado anteriormente.

Apesar de não estar relacionado diretamente com a hipotese da otimização, foi feita uma medição de *speed-up* para este *dataflow*. O resultado aparece na Figura 33-(b) onde observa-se um fato curioso: O ganho no uso dos recursos do *cluster* é inferior no caso de gravação de *dataset* menor (Clientes). Este é um comportamento recorrente que foi observado em todas as execuções executadas no Cluster.



(a) Gráfico com o tempo decorrido para `W0`

(b) Variação do *speed-up* para o *dataflow* `W0`

Figura 33: Gráfico com o tempo decorrido na execução do *dataflow* `W0` que usa os dois *datasets* (*customer* e *order*) e variação do *speed-up* para o *dataflow*

4.6.2 Análise da otimização de UDF com Catalyst (`W1A` e `W1B`)

Com os *datasets* já no formato adequado, após a execução do *workflow* `W0`, executou-se o *workflow* `W1A` que equivale ao *dataflow* da *query* `q13` do TPC-H, **sem otimização**, nos dois operadores *wFilter* contíguos com UDF. Este *dataflow* foi executado sobre o grupo *short* de *datasets* com a primeira UDF filtrando quantidades de pedidos equivalente a aproximadamente 90% do total. Os dois operadores *wFilter* contíguos com UDF possuem custos de execução consideravelmente diferentes. Ele foi executado com uma quantidade limitada de registros de entrada (grupo *short*) de forma a executar em tempo hábil no *cluster* de 24 cores do CEFET. Os filtros `remoteDoNotContains(o_comment, '1%')` da atividade **`A2`** e `doNotContains(o_comment, '2%')` da atividade **`A3`** são executados em sequência permitindo otimização pelo *Catalyst* e, assim, o código gerado no *Spark* transforma os dois operadores em um único, usando operador lógico AND entre os dois predicados. Esta otimização é aquela da Regra 1 que é intrínseca ao *Spark* e é executada pelo *Catalyst* após a otimização provida pelo WfF.

Tabela 5: Tabela com o tempo decorrido, o speed-up e a eficiência para diversas configurações de quantidade de *cores* relativo ao *dataflow* `W1A`

#	A	B	SupA	SupB	ξA	ξB
1	6645	6502	1.00	1.00	1.00	1.00
2	3398	3404	1.96	1.91	0.98	0.96
4	1883	1953	3.53	3.33	0.88	0.83
6	1142	1129	5.82	5.76	0.97	0.96
8	1332	1330	4.99	4.89	0.62	0.61
10	756	759	8.79	8.57	0.88	0.86
12	757	762	8.78	8.53	0.73	0.71
14	974	960	6.82	6.77	0.49	0.48
16	754	757	8.81	8.59	0.55	0.54
18	392	391	16.95	16.63	0.94	0.92
20	394	391	16.87	16.63	0.84	0.83
22	397	389	16.74	16.71	0.76	0.76
24	415	387	16.01	16.80	0.67	0.70

Execução do *dataflow* `W1A`

Os tempos de execução para o *dataflow* `W1A` foram registrados para a versão sem otimização de filtros contíguos contendo UDF e aparecem na Tabela 5.

A primeira coluna da esquerda apresenta o número de cores utilizado. Foram feitas duas execuções nas mesmas condições para que se eliminasse possíveis dúvidas sobre o comportamento do *dataflow* no *cluster*, desta forma, a coluna A apresenta os valores de tempo decorrido, medido em segundos, para a primeira execução e a coluna B apre-

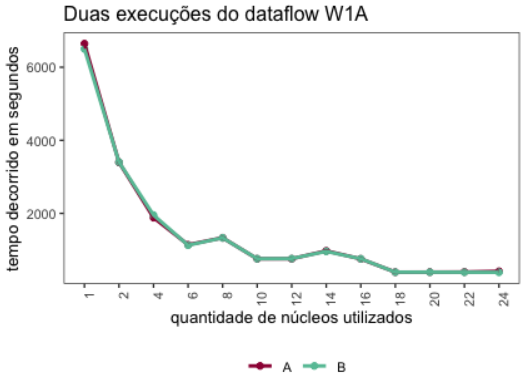
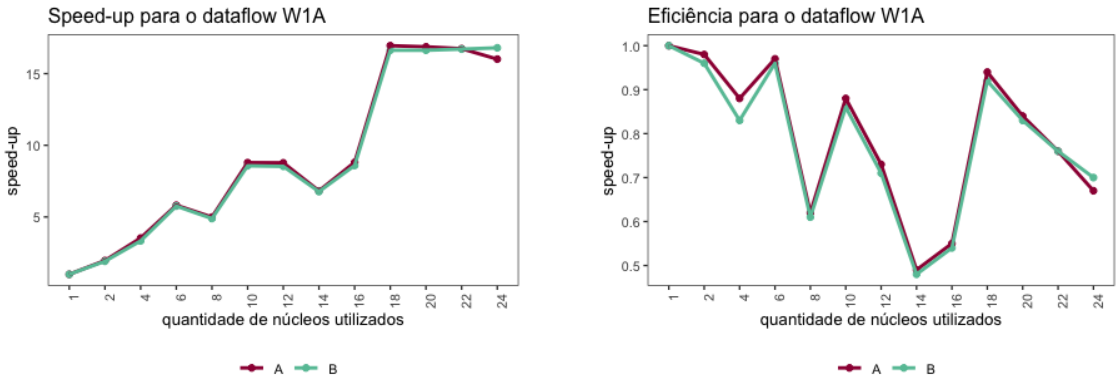


Figura 34: Gráfico com o tempo decorrido na execução do *dataflow* `W1A` que usa os dois *datasets* (*customer* e *order*). O *dataflow* foi executado duas vezes para verificar se o comportamento anômalo persistia.

senta os valores de tempo decorrido, também medido em segundos, para a segunda execução. As colunas SupA e SupB apresentam os valores para *speed-up* da primeira e segunda execução, respectivamente. Por último, as colunas ξA e ξB apresentam as eficiências calculadas, usando a Expressão 8 sobre as medidas de *speed-up*.



(a) Variação do *speed-up* para `W1A`

(b) Variação da eficiência para `W1A`

Figura 35: Variação do *speed-up* e eficiência relativo a execução do *dataflow* `W1A`

O gráfico de tempo decorrido aparece na Figura 34 onde pode ser observado um comportamento anômalo [8] para as execuções do com 8, 14 e 16 cores. O *speed-up* está ilustrado na Figura 35-(a) que deixa mais evidente a anomalia citada. O gráfico de eficiência aparece na Figura 35-(b) que mostra uma variança grande devido a anomalia, porém melhores que nos *dataflows* anteriores se considerarmos a média.

[8] Diz-se anômalo, pois a curva do *speed-up* não segue o padrão esperado e esse comportamento será discutido posteriormente na análise da execução do *dataflow* `W1B`

Execução do *dataflow* `W1B`

Os tempos de execução foram registrados também para a versão com otimização de filtros contíguos contendo UDF e aparecem na tabela 6.

Tabela 6: Tabela com o tempo decorrido, o speed-up e a eficiência para diversas configurações de quantidade de *cores* relativo ao *dataflow* `W1B`

#	A	B	SupA	SupB	ξA	ξB
1	5503	5618	1.00	1.00	1.00	1.00
2	2809	2866	1.96	1.96	0.98	0.98
4	1652	1603	3.33	3.50	0.83	0.88
6	1033	974	5.33	5.77	0.89	0.96
8	1130	1135	4.87	4.95	0.61	0.62
10	652	649	8.44	8.66	0.84	0.87
12	647	650	8.51	8.64	0.71	0.72
14	818	820	6.73	6.85	0.48	0.49
16	645	645	8.53	8.71	0.53	0.54
18	355	332	15.50	16.92	0.86	0.94
20	333	333	16.53	16.87	0.83	0.84
22	334	334	16.48	16.82	0.75	0.76
24	333	333	16.53	16.87	0.69	0.70

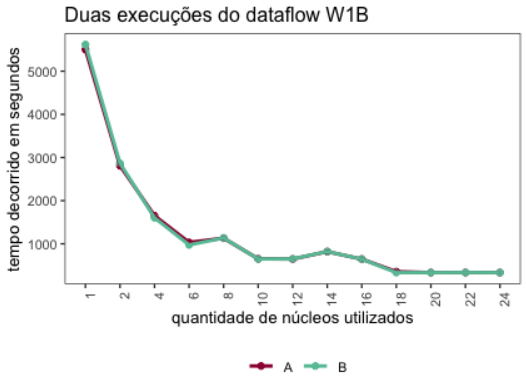
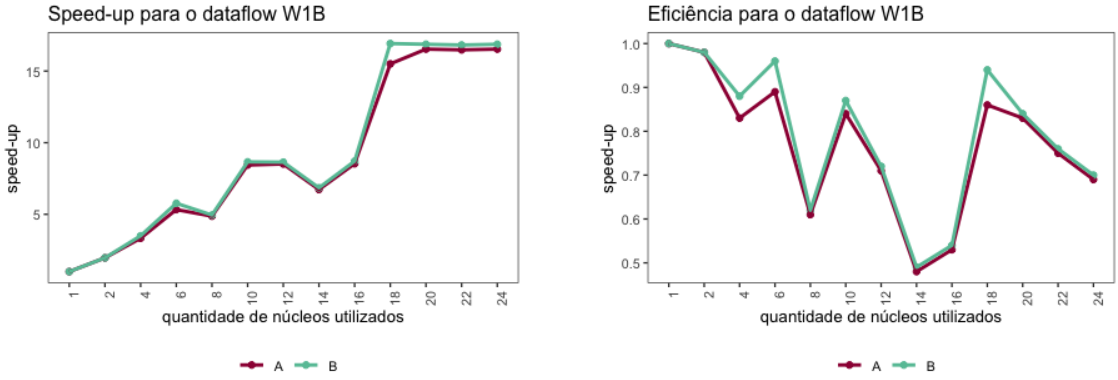


Figura 36: Gráfico com o tempo decorrido na execução do *dataflow* `W1B`. O *dataflow* foi executado duas vezes (A e B) para verificar se o comportamento anômalo persistia.

O gráfico de tempo decorrido aparece na Figura 36. O *speed-up* está ilustrado na Figura 37-(a) e o gráfico de eficiência aparece na Figura 37-(b).

Pode ser observado nos gráficos que a execução do *dataflow* `W1A` e `W1B` apresentou uma curva de *speed-up* fora do padrão, com desvios relevantes para 8, 14 e 16 cores. Para avaliar este comportamento executou-se outro experimentos onde foi usado um *dataset* cem vezes maior (*large*) e os resultados aparecem nas Figuras 38 (a) e (b)

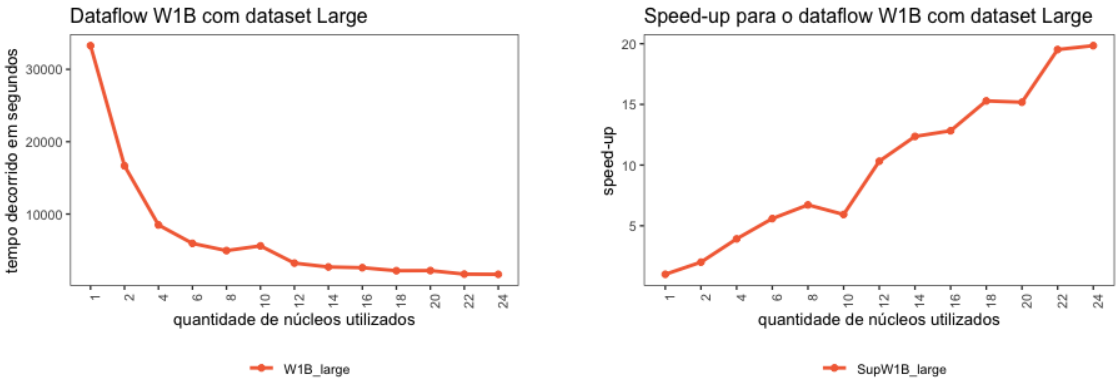
Como nos outros casos, observa-se que ao aumentar a cardinalidade dos *datasets* o paralelismo apresenta melhor eficiência no *cluster*. Comparando a versão otimizada com



(a) Variação do *speed-up* para `W1B`

(b) Variação da eficiência para `W1B`

Figura 37: Variação do *speed-up* e eficiência relativo a execução do *dataflow* `W1B`



(a) Tempo decorrido para `W1B` - large

(b) *Speed-up* para `W1B` - large

Figura 38: Variação do tempo decorrido e do *speed-up* para o *dataflow* `W1B` usando o conjunto de *dataset large*

a não otimizada observa-se um ganho de aproximadamente 15% na versão otimizada, em termos de tempo decorrido nas execuções. Para analisar a vantagem obtida no WfF com a otimização nos operadores *wFilter* contíguos com UDF, foi feito também um estudo mais rigoroso, utilizando o projeto *sparkMeasure*, disponível no GitHub, que permite coletar métricas detalhadas a nível de estágio (*stage*) e a nível de tarefa (*task*) que é mais detalhada e de granularidade fina. Para o objetivo dessa dissertação, nesta avaliação experimental, basta as métricas a nível de *stage* (de maior granularidade), para que se tire as conclusões desejadas. As métricas a nível de *task* podem ser avaliadas em trabalhos futuros.

Além das medidas de tempo decorrido obtidas pelo código gerado pelo *framework* WfF utilizou-se a ferramenta *sparkMeasure* para coletar métricas mais detalhadas a ní-

vel de estágio (*Stage*) do *Spark*. São métricas de granularidade alta que ao serem tomadas não impactam o processo em produção de forma significativa. A Tabela 7 mostra diversas medidas obtidas com o *sparkMeasure* que são métricas a nível de *stage* que fornece o número de estágios, a quantidade de *tasks* alocadas em todos os *workers* para cada instância de execução além do tempo gasto com *Garbage Collector* (GC). As medidas mostradas na Tabela 7 são apenas algumas das disponíveis na ferramenta. Optou-se por não considerar nesta análise experimental as medidas de consumo de I/O, por exemplo. Como o *sparkMeasure* usa a API *SparkListener* do *Apache Spark* ele pode ser integrado no *framework* WfF, e poderá ser objeto de trabalho futuro de forma que que se tenha um repositório de proveniência retrospectiva bastante rico para apoiar heurísticas de otimização de *dataflow*.

A tabela 7, obtida com o *sparkMeasure*, ilustra as medidas tomadas com 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22 e 24 cores. Como esperado, foi observado um ganho aproximado de 15% com a otimização desenvolvida no WfF e usada no *dataflow* para a *query* `q13` do TPC-H. Os dois *tokens* que caracterizam o pedido especial nesta *query* são: *pending* e *requests* mas o resultado é semelhante para outras escolhas de *tokens*. O ganho da otimização é obtido pois uma das UDF possui maior custo de execução por ser executada remotamente usando WebService REST e a outra é executada diretamente na JVM e seu custo de execução é aproximadamente 4 milissegundos menor por cada tupla processada pela UDF.

Uma análise mais cuidadosa mostra que: dos 1.500.000 pedidos existentes no *dataset order*, 1.360.273 não possuem o *token pending* e desses 1.157.874 também não possuem o *token requests*. Uma das UDF filtra os pedidos que não contém *pending* e a outra filtra os pedidos que não contém *requests* e como uma delas executa remotamente, análogo a *Remote Procedure Call* (RPC), apresenta resultado com maior custo de execução. A otimização testada força a execução antecipada da UDF de menor custo de processamento que é aplicada sobre 1.500.000 tuplas e desta forma, a UDF de maior custo de processamento executa sobre apenas 1.360.273 tuplas. O resultado final são 1.157.874 tuplas de pedidos que permanecem para processamentos posteriores.

Aqui vale uma avaliação sobre o efeito da seletividade do predicado. Observe que apenas 9.3%das tuplas são responsáveis por uma melhoria de 14,75%

Vale lembrar que os dados que orientam a otimização devem vir de um repositório de proveniência retrospectiva, como foi discutido na Seção 1.6. Na próxima Seção

dataflow	cores	# stages	# tasks	decorrido	GC	sem GC	ganho %
W1A	1	4	404	6638	19	6616	
W1B	1	4	404	5495	18	5472	17,2
W1A	2	4	404	3391	18	6573	
W1B	2	4	404	2796	17	5425	17,5
W1A	4	4	407	1873	11	6547	
W1B	4	4	407	1641	12	5683	12,3
W1A	6	4	412	1133	9	6505	
W1B	6	4	412	1025	9	5946	9,6
W1A	8	4	415	1324	16	6443	
W1B	8	4	415	1123	15	5487	15,2
W1A	10	4	415	746	13	6427	
W1B	10	4	415	643	12	5481	13,8
W1A	12	4	415	747	14	6430	
W1B	12	4	415	638	12	5473	14,6
W1A	14	4	415	950	24	6441	
W1B	14	4	415	809	19	5478	14,8
W1A	16	4	404	746	26	6470	
W1B	16	4	404	637	23	5516	14,6
W1A	18	4	404	383	30	6497	
W1B	18	4	404	345	28	5613	9,8
W1A	20	4	424	384	29	6504	
W1B	20	4	424	323	24	5515	15,9
W1A	22	4	424	388	28	6532	
W1B	22	4	424	325	27	5536	16,2
W1A	24	4	424	406	27	6545	
W1B	24	4	424	324	25	5519	20,1

Tabela 7: Tabela obtida com o *sparkMeassure* mostra o comparativo de ganho devido a otimização proposta para filtros contíguos usando UDF e semântica anotada pelo *framework*. Foi obtido um ganho médio percentual de 14,75% em termos de tempo decorrido.

apresenta-se o comportamento do *cluster* em termos de distribuição do processamento, a medida que aloca-se mais cores para processamento dos *dataflows*.

4.7 Diferença de abordagens e a estratégia de otimização do *Spark*.

É preciso deixar claro que a API `Dataset` do *Spark*, por ser fortemente tipada, obriga que cada relação de saída intermediária tenha uma classe que implemente o *Design Pattern ValueObject* para estruturar o dado e passá-lo adiante. A API `DataFrame`, por outro lado, só precisa de uma lista de atributos, cada um com um nome arbitrário e um tipo. Por este motivo o `DataFrame` é mais flexível, já que esta lista de atributos é estruturada em um tipo genérico chamado Row. A tipagem neste caso é um pouco mais frágil, porém o otimizador *Catalyst* tem mais liberdade para fazer otimizações com operador Project, pois não fica restrito a gerar os objetos intermediários em sua definição rígida.

Existe uma alternativa para não abrir mão da tipagem forte da API `Dataset` e ao

mesmo tempo obter código otimizado, mas neste caso as otimizações relacionadas ao operador Project devem ser feitas no *framework* WfF antes de gerar as definições dos *ValueObject*. Ou seja, ao final da etapa de construção do grafo, uma análise adicional deve ser feita caminhando no grafo de forma reversa anotando os atributos que não são necessários a execução da consulta e desta forma poder gerar o código já considerando esta informação. Esta abordagem permite aliar os benefícios de um código fortemente tipado, onde os erros lógicos poderão ser reconhecidos prematuramente ainda na fase de compilação do *dataflow*, sem perder oportunidades de otimizar operações relacionais do tipo Project. Esta situação pode ser observada no código gerado para o *dataflow* de exemplo da *query* `q13` do TPC-H. Sem essa referida otimização do WfF o esquema (*schema*) de R3 será dado pelo *ValueObject* R3_Tuple que possui nove (9) atributos.

```
case class R3_Tuple(o_orderkey: Int, x_custkey: Int, o_orderstatus: String,
o_totalprice: Double, o_orderdate: Int, o_orderpriority: String, o_clerk: String,
o_shippriority: Integer, o_comment: String)
```

Uma análise mais cuidadosa do grafo até o final antes de gerar o código verificaria que basta apenas dois (2) atributos em R3_Tuple, pois apenas estes são usados adiante.

```
case class R3_Tuple(x_custkey: Int, o_comment: String)
```

Esta otimização mencionada pode ser objeto de trabalhos futuros. Na próxima Seção apresenta-se um questionamento sobre a ameaça a validade e em seguida uma discussão sobre a avaliação experimental propondo as interpretações para os resultados dos experimentos realizados.

4.8 Ameaça a validade

A pesquisa cientifica quantitativa é baseada na medida numérica de algumas poucas variáveis objetivas [9] com ênfase na comparação de resultados e no uso de técnicas estatísticas.

Em termos de validade de experimento para esta avaliação experimental, considera-se dois tipos principais, que são: *i)* interna e *ii)* externa [Wainer et al., 2007]. Dizemos que um experimento tem boa validade interna se os valores medidos obtidos são realmente devidos à manipulação da variável objetiva (e não decorrentes de um outro fator).

[9] Variáveis a serem observadas são consideradas objetivas quando diferentes observadores obterão os mesmos resultados em observações distintas, considerando uma margem de erro aceitável. Além disso existe um consenso na comunidade científica sobre o que é melhor e o que é pior para os valores dessas variáveis objetivas

Dizemos que um experimento tem boa validade externa se os resultados são generalizáveis, ou seja, não são únicos para um conjunto particular de dado e processo de transformação.

No nosso experimento de validação da hipótese de otimização do *dataflow* baseado na *query* `q13`, não são observadas ameaças à validade interna já que a infraestrutura computacional garante as condições definidas no experimento: *i)* escolha da versão otimizada versus a não otimizada `W1A` X `W1B`; *ii)* escolha da quantidade de cores e *iii)* medidas do tempo decorrido via instrumentação.

Por outro lado podemos observar ameaças a validade do tipo externa, pois foi usada apenas uma *query*. Podemos justificar que a *query* `q13`, tem um baixo índice de seletividade por usar uma negação nos predicados dos filtros (seleciona tuplas que NÃO contém um dado token) e que mesmo nestas condições foram observados resultados de 15% de otimização. De qualquer forma sabe-se que a maneira eficaz de demonstrar capacidade de generalização é repetir o experimento muitas vezes, em muitas situações diferentes. Assim, trabalhos futuros poderão avaliar outras condições de predicado e outras *queries* envolvendo mais de uma UDF em filtros contíguos, para validar com mais robustez a hipótese de otimização apresentada neste trabalho. Além disso, trabalhos futuros poderão avaliar otimizações de ponta a ponta em *dataflow* contendo filtros e junções como analisado por Hellerstein [1992].

4.9 Discussão sobre a avaliação experimental

O *cluster* que foi usado para teste apresentou um comportamento de deterioração de desempenho para um certo número de cores quando estes eram escolhidos como 8, 14 e 16. A Tabela 8 mostra os valores de tempo decorrido para 5 execuções do *dataflow* `W1A` (sem otimização).

A Tabela 9 mostra os valores de tempo decorrido para 5 execuções do *dataflow* `W1B` (com otimização).

É possível verificar com estas duas tabelas que em todas as execuções este comportamento se manteve, já que o desvio padrão ficou abaixo de 8%, ou seja, trata-se de uma característica intrínseca ao próprio *cluster* e o ao gerenciador de escalonamento implementado pelo *Spark*. Seria necessário um estudo mais aprofundado das métricas de uso de I/O, Threads, CPU, memória e distribuição de *Tasks* para explicar este comportamento, mas isso está fora do escopo desta avaliação experimental e poderá ser

# cores	W1A A	W1A B	W1A C	W1A D	W1A E	W1A μ	W1A σ	W1A σ %
1	6.638	6.494	6.624	6.444	6.656	6.571	95,8	1,5
2	3.391	3.391	3.389	3.473	3.392	3.407	36,5	1,1
4	1.873	1.942	1.871	1.891	1.882	1.892	29,2	1,5
6	1.133	1.121	1.128	1.126	1.131	1.128	4,6	0,4
8	1.322	1.321	1.324	1.322	1.322	1.322	1,2	0,1
10	746	752	753	754	755	752	3,5	0,5
12	747	754	752	752	752	752	2,8	0,4
14	965	951	960	961	936	954	11,6	1,2
16	745	748	744	744	684	733	27,2	3,7
18	383	381	378	384	372	380	4,8	1,3
20	384	381	378	382	399	385	8,2	2,1
22	388	380	380	396	411	391	13,0	3,3
24	406	380	379	380	424	394	20,1	5,1

Tabela 8: Valores de tempo decorrido, em segundos, para cinco execuções (A, B, C. D e E) do *dataflow* W1A (sem otimização). A coluna W1A μ apresenta a média e a coluna W1A σ apresenta o desvio padrão dos tempos decorridos. O valor percentual do desvio padrão aparece na coluna W1A σ %.

# cores	W1B A	W1B B	W1B C	W1B D	W1B E	W1B μ	W1B σ	W1B σ %
1	5.495	5.611	5.622	5.652	5.742	5.624	88,7	1,6
2	2.796	2.859	2.879	2.871	2.870	2.855	33,8	1,2
4	1.641	1.596	1.591	1.604	1.593	1.605	20,9	1,3
6	1.025	965	960	966	967	977	26,9	2,8
8	1.123	1.126	1.127	1.125	1.121	1.124	2,4	0,2
10	643	639	643	642	636	641	2,9	0,5
12	638	643	642	642	644	642	2,4	0,4
14	809	812	816	818	964	844	67,2	7,9
16	637	635	636	636	635	636	0,9	0,1
18	345	323	324	325	333	330	9,4	2,8
20	323	324	323	331	324	325	3,4	1,0
22	325	324	325	324	323	324	0,9	0,3
24	324	324	323	325	323	324	1,0	0,3

Tabela 9: Valores de tempo decorrido, em segundos, para cinco execuções (A, B, C. D e E) do *dataflow* W1B (com otimização). A coluna W1B μ apresenta a média e a coluna W1B σ apresenta o desvio padrão dos tempos decorridos. O valor percentual do desvio padrão aparece na coluna W1B σ %.

tema de trabalhos futuros.

A Figura-39 expressa este comportamento de forma gráfica e facilita a sua visualização.

Calculou-se também a regressão polinomial de grau 3 para os *dataflows* (usando a biblioteca Python Numpi). O comportamento foi o mesmo e são ilustrados na Figura 40-(a), junto aos valores obtidos em uma das execuções do *dataflow* W1A.

Regressão polinomial de grau 3. Coeficientes:

$a = -2.071,70$

$b = 94.683,87$

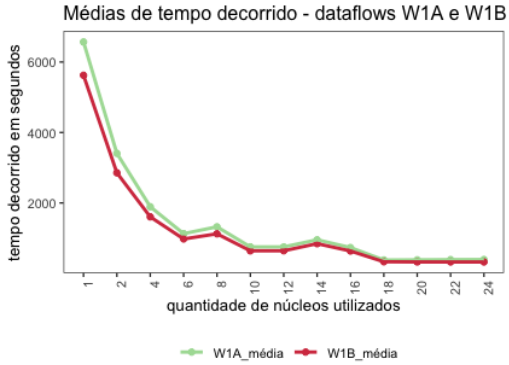


Figura 39: Comparando as medidas de tempo decorrido para dez execuções, sendo cinco para o *dataflow* `W1A` (sem otimização) e cinco para o *dataflow* `W1B` (com otimização). O gráfico apresenta os valores médios de cada *dataflow*.

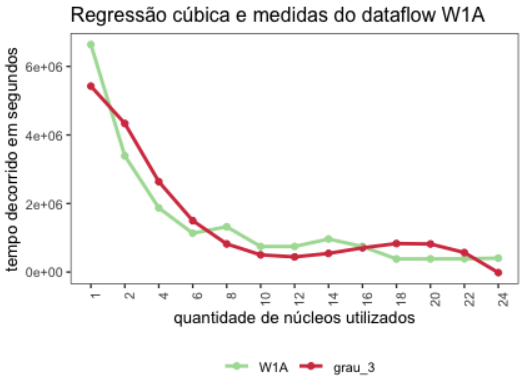
$c = -1.358.565,40$

$d = 6.690.570,90$

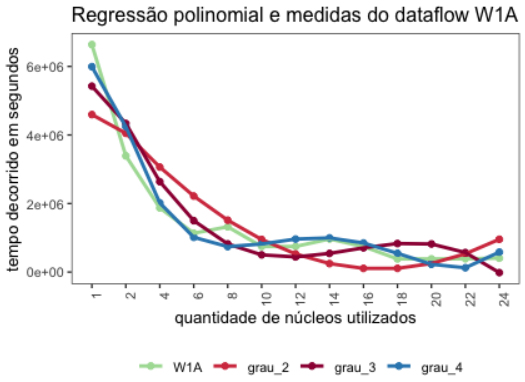
com $y = ax3 + bx2 + cx + d$

A Figura 40-(a) mostra a comparação entre as medidas de tempo decorrido e os pontos obtidos por regressão polinomial de ordem 3 para o *dataflow* `W1A`.

Veja que quando número de cores ultrapassa 12 a curva de interpolação passa a ter derivada positiva indicando que começa a ficar ruim o desempenho no *cluster*, ou seja, achar o mínimo do polinômio de grau três obtido na regressão polinomial permite encontrar o número máximo de cores que devem ser usados para um dado *dataflow*, e isso pode enriquecer o repositório de proveniência retrospectiva.



(a) Comparação entre tempo decorrido e regressão cúbica para o *dataflow* `W1A`



(b) Regressão polinomial de ordem 2, 3 e 4 para o *dataflow* `W1A`

Figura 40: Comparação entre as medidas de tempo decorrido e os pontos obtidos por regressão polinomial de ordem 2, 3 e 4 para o *dataflow* `W1A`

Usando regressões polinomiais de maior ordem apresentam coeficientes de determinação melhores porém passa a ocorrer o fenômeno de sobreajuste (*overfitting*) [10]. Particularmente para o caso de determinar a quantidade recomendada de cores a ser usada em um *dataflow*, o polinômio de grau 3 produziu melhor resultado. Para ilustrar veja a Figura 40-(b) comparando três regressões polinomiais para o *dataflow* `W1A`.

Os coeficientes de determinação, que medem o quanto o polinômio se aproxima dos dados reais (quanto mais próximo de 1 significa que melhor se aproxima), para estas regressões mostradas, são os seguintes:

```
Polinômio de ordem 2:  0.767
Polinômio de ordem 3:  0.885
Polinômio de ordem 4:  0.954
```

Assim encerra-se a análise da avaliação experimental e no próximo capítulo apresenta-se as considerações finais sobre esta dissertação.

[10] *overfitting* ocorre quando o modelo se ajusta muito bem a um dado conjunto de dados, mas é ineficaz para prever novos resultados a partir de novos dados

(Página intencionalmente deixada em branco)

Considerações Finais

Este trabalho propõe a criação de um *framework* (WfF) para análise de dados em larga escala em ambiente distribuído com escalabilidade horizontal (DISC), mais especificamente no *Apache Spark*, provendo um modelo independente de computação (CIM), independente de plataforma (PIM) e específico de plataforma (PSM), como propostos na arquitetura MDA, definindo uma linguagem agnóstica no formato de uma eDSL em Kotlin para especificação dos *workflows* pelos cientistas e engenheiros de dados. Tal linguagem deve permitir a especificação de UDF regidas por operadores algébricos, compatíveis com a álgebra de *workflow* de Ogasawara et al. [2011]. Além disso, o *framework* deve aceitar UDF escrita em linguagem Python, R, Java, Scala e Kotlin como proposto em [Ferreira et al., 2017] e permitir, de forma parametrizada, a inclusão de otimização de *dataflow* usando a álgebra de *workflow* valendo se de informações de proveniência retrospectiva colhida do *Apache Spark* via *SparkListener*.

Contribuição

Com a implementação deste *framework*, verificou-se que é possível criar otimizações algébricas incluindo UDF com o uso dos pontos de extensão disponíveis no *Spark SQL* e *Catalyst*. As otimizações propostas neste trabalho são largamente utilizadas em banco de dados e podem também ser usadas em *workflows* de análise de dados. Os experimentos foram realizados em ambiente de computação distribuída para validar as hipóteses de otimização algébrica. Em relação a otimização de operações *wFilter* sobre UDF usando o *dataset* do *benchmark* TPC-H foi obtido um ganho médio de 14,75 %, em termos de tempo total de execução.

Uma contribuição marginal é a capacidade de transparência em relação a fonte de dado inspirada na MDA. Uma prova de conceito foi realizada para validar o requisito de independência de plataforma e a transparência na escolha de fontes de dado (*datasource*) onde se chegou a um resultado satisfatório.

Limitações

Dado que o escopo do trabalho se revelou muito amplo para o tempo disponível, fez-se inicialmente uma análise bastante ampla e dividiu-se o projeto em fases. Alguns recortes foram feitos para que se chegasse a uma solução adequada no tempo desejado. Neste recorte, envolvendo a implementação, a linguagem agnóstica foi definida apenas para alguns operadores da álgebra de *workflow*. A geração de código foi implementada apenas para a linguagem Scala, e a proveniência foi considerada como fornecida pelo sistema externo.

É desejável que algumas implementações sejam realizadas e agregadas ao WfF. São elas: *i)* implementar os outros operadores da álgebra de *workflow* e as respectivas heurísticas de otimização; *ii)* prover a integração com o ambiente de *runtime* do *Apache Spark* via *SparkListener* para gerar informações de proveniência retrospectiva e realimentar o sistema; *iii)* estender o WfF implementando a geração de código para as linguagens R e Python permitindo a execução nos ambiente *Spark R* e *PySpark*; *iv)* estender o WfF implementando um *wrapper* do *ExternalDriver* para código em linguagem Python permitindo integração com Pandas, TensorFlow e Plotnine [Kibirige et al., 2018], usando a base descrita na Seção 3.3; *v)* melhorar a implementação do *wrapper* do *ExternalDriver* para a linguagem R (classe *RDriver*) de forma que permita a definição de UDAF em R usando os pacotes ARIMA, TSPred, STMotif e a submissão de *dataflow* do WfF pelo ambiente R / RStudio; *vi)* codificar regras semânticas da eDSL para melhorar o tratamento de erro e o feedback ao usuário; *vii)* avaliar e testar o efeito do particionamento dos *datasets* na otimização quando se usa o YARN e o Kubernetes; *viii)* coletar métricas detalhadas com *sparkMeasure* a nível de estágio (*stage*) e a nível de tarefa (*task*) de granularidade fina

Trabalhos futuros

A implementação de alguns dos requisitos funcionais do *framework* são propostos para trabalhos futuros. São eles: *i)* adaptar outras heurísticas propostas por Ogasawara [2011] para este ambiente provido pelo ecossistema *SparkSQL*/*Hadoop*; *ii)* analisar possíveis abordagens de implementação no *Apache Spark* para as estratégias de despacho (*Dispatch*) [Ogasawara, 2011], que trata da execução de UDF regida por operadores em um fluxo de dados; *iii)* otimizar operações relacionais do tipo *Project* no grafo do *dataflow*

no WfF, ou seja antes de submeter ao *Catalyst*, para ter os benefícios da tipagem forte da API `Dataset` e ao mesmo tempo obter código otimizado; *iv)* avaliar outras condições de predicado e outras *queries* envolvendo mais de uma UDF em filtros contíguos, para validar com mais robustez a hipótese de otimização apresentada neste trabalho; *v)* avaliar otimizações de ponta a ponta em *dataflow* contendo filtros e junções como analisado por Hellerstein [1992]

(Página intencionalmente deixada em branco)

Referências Bibliográficas

Alexandrov, A., Kunft, A., Katsifodimos, A., Schüler, F., Thamsen, L., Kao, O., Herb, T., and Markl, V. (2015). **Implicit parallelism through deep language embedding.** In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 47–61. ACM. 63

Alexandrov, A., Salzmann, A., Krastev, G., Katsifodimos, A., and Markl, V. (2016). **Emma in action: Declarative dataflows for scalable data analysis.** In *Proceedings of the 2016 International Conference on Management of Data*, pages 2073–2076. ACM. 23

Almeida, A. M., Lima, M. I., Macedo, J. A., and Machado, J. C. (2016). **Dmm: A distributed map-matching algorithm using the mapreduce paradigm.** In *Intelligent Transportation Systems (ITSC), 2016 IEEE 19th International Conference on*, pages 1706–1711. IEEE. 29

Armbrust, M., Xin, R., Lian, C., Huai, Y., Liu, D., Bradley, J., Meng, X., Kaftan, T., Frankliny, M., Ghodsi, A., and Zaharia, M. (2015). **Spark SQL: Relational data processing in spark.** In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 2015-May, pages 1383–1394. 20, 29, 32, 60

Brown, A. W. (2004). **Model driven architecture: Principles and practice.** *Software and Systems Modeling*, 3(4):314–327. 53

Bryant, R. E. (2011). **Data-intensive scalable computing for scientific applications.** *Computing in Science & Engineering*, 13(6):25–33. 19, 28

Callahan, S. P., Freire, J., Santos, E., Scheidegger, C. E., Silva, C. T., and Vo, H. T. (2006). **Vistrails: visualization meets data management.** In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 745–747. ACM. 61

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). **Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems.** *arXiv preprint arXiv:1512.01274*. 62

Cheney, J., Lindley, S., and Wadler, P. (2013). **A practical theory of language-integrated query.** *ACM SIGPLAN Notices*, 48(9):403–416. 32

Chirigati, F., Capone, R., Shasha, D., Rampin, R., and Freire, J. (2017). **A collaborative approach to computational reproducibility.** *arXiv preprint arXiv:1709.01154*. 49

Chronis, Y., Foufoulas, Y., Nikolopoulos, V., Papadopoulos, A., Stamatogiannakis, L., Svingos, C., and Ioannidis, Y. E. (2016). **A relational approach to complex dataflows.** In *EDBT/ICDT Workshops*. 27

Dave, A., Jindal, A., Li, E., Xin, R., Gonzalez, J., and Zaharia, M. (2016). **GraphFrames: An integrated API for mixing graph and relational queries.** In *ACM International Conference Proceeding Series*, volume 24-June-2016. cited By 0. 30

Dean, J. and Ghemawat, S. (2008). **MapReduce: Simplified data processing on large clusters.** *Communications of the ACM*, 51(1):107–113. 19

Deng, S., Huang, L., Taheri, J., and Zomaya, A. Y. (2015). **Computation offloading for service workflow in mobile cloud computing.** *IEEE Transactions on Parallel and Distributed Systems*, 26(12):3317–3329. 23

Dijkstra, E. W. (1982). **On the role of scientific thought.** In *Selected writings on computing: a personal perspective*, pages 60–66. Springer. 23

Dumbill, E. (2005). **Ruby on rails: An interview with david heinemeier hansson.** Technical report, O'Reilly Media, Inc. 23

Elmasri, R. and Navathe, S. (2015). ***Fundamentals of database systems*.** Pearson Education India. 33, 34, 45, 47, 48

Felleisen, M., Findler, R. B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., and Tobin-Hochstadt, S. (2018). **A programmable programming language.** *Communications of the ACM*, 61(3):62–71. 23

Ferreira, J., Gaspar, D., Monteiro, B., Silva, A. B., Porto, F., and Ogasawara, E. (2017). **Uma Proposta de Implementação de Álgebra de Workflows em Apache Spark no Apoio a Processos de Análise de Dados.** In *Brazilian e-Science Workshop*. 22, 67, 85, 113

Frankel, D. S. (2003). ***Model driven architecture applying MDA.*** John Wiley & Sons. 21, 53

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). **Design patterns: elements of.** 54

Gantz, J. and Reinsel, D. (2011). **Extracting value from chaos.** *IDC iview*, 1142(2011):1–12. 19

Graefe, G. (1994). **Volcano/spl minus/an extensible and parallel query evaluation system.** *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135. 35

Hellerstein, J. M. (1992). **Predicate migration: Optimizing queries with.** *ACM*. 24, 108, 115

Hsu, M., Chen, Q., Wu, R., Zhang, . B., and Zeller, H. (2010). **Generalized udf for analytics inside database engine.** In *International Conference on Web-Age Information Management*, pages 742–754. Springer. 40, 47

Hwang, K. and Xu, Z. (1998). ***Scalable parallel computing: technology, architecture, programming.*** McGraw-Hill, Inc. 92

Jemerov, D. and Isakova, S. (2017). ***Kotlin in Action.*** Manning. 23

Kibirige, H., Lamp, G., Katins, J., O., A., gdowding, Funnell, T., matthias k, Arnfred, J., Blanchard, D., Chiang, E., Astanin, S., Kishimoto, P. N., Sheehan, E., stonebig, Gibboni, R., Willers, B., Pavel, Halchenko, Y., smutch, zachcp, Collins, J., RK, M., Wickham, H., guoci, Brian, D., Arora, D., Brown, D., Becker, D., Koopman, B., and Anthony (2018). **has2k1/plotnine: v0.5.1.** 114

Kumar, V., Grama, A., Gupta, A., and Karypis, G. (1994). ***Introduction to parallel computing: design and analysis of algorithms***, volume 400. Benjamin/Cummings Redwood City. 92

Matsunaga, A., Tsugawa, M., and Fortes, J. (2008). **Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications.** In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 222–229. IEEE. 61

McClatchey, R. (2018). **The deployment of an enhanced model-driven architecture for business process management.** *arXiv preprint arXiv:1803.07435*. 53

Meng, X., Bradley, J., Yuvaz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. (2016). **Mllib: Machine learning in apache spark.** *JMLR*, 17(34):1–7. 29, 63

Meyer, B. (1988). ***Object-oriented software construction***, volume 2. Prentice hall New York. 68

Morcel, R., Ezzeddine, M., and Akkary, H. (2016). **FPGA-Based Accelerator for Deep Convolutional Neural Networks for the SPARK Environment.** In *Proceedings - 2016 IEEE International Conference on Smart Cloud, SmartCloud 2016*, pages 126–133. 19

Neumann, T. (2011). **Efficiently compiling efficient query plans for modern hardware.** *Proceedings of the VLDB Endowment*, 4(9):539–550. 37

Nothaft, F., Massie, M., Danford, T., Zhang, Z., Laserson, U., Yeksigian, C., Kottalam, J., Ahuja, A., Hammerbacher, J., Linderman, M., Franklin, M., Joseph, A., and Patterson, D. (2015). **Rethinking data-intensive science using scalable analytics systems.** In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 2015-May, pages 631–646. 19

Ogasawara, E., de Oliveira, D., Valduriez, P., Dias, J., Porto, F., and Mattoso, M. (2011). **An algebraic approach for data-centric scientific workflows.** In *Proceedings of the VLDB Endowment*, volume 4, pages 1328–1339. 20, 28, 46, 47, 48, 67, 113

Ogasawara, E. S. (2011). **Tech report: Tese de doutorado - uma abordagem algébrica para workflows científicos com dados em larga escala.** Technical Report 12/2011, Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE / Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brazil, Ilha do Fundão, Rio de Janeiro, Brazil. 46, 59, 60, 61, 114

Pattis, R. E. (2013). **Ebnf: A notation to describe syntax.** 56

Rheinländer, A., Heise, A., Hueske, F., Leser, U., and Naumann, F. (2015). **Sofa: An extensible logical optimizer for udf-heavy data flows.** *Information Systems*, 52:96–125. 61

Rheinländer, A., Leser, U., and Graefe, G. (2017). **Optimization of complex dataflows with user-defined functions.** *ACM Computing Surveys*, 50(3). 20, 40, 41, 46

Santos, T. G. (2018). **Análise e captura de dados de proveniência no apache spark.** Technical Report 01/2018, Department of computer science, UFF/Brazil, Niteroi, Rio de Janeiro, Brazil. 50

Silberschatz, A., Korth, H. F., Sudarshan, S., et al. (2011). ***Database system concepts*.** McGraw-Hill New York. 44

Singh, Y. and Sood, M. (2009). **Model driven architecture: A perspective.** In *Advance Computing Conference, 2009. IACC 2009. IEEE International*, pages 1644–1652. IEEE. 21, 53

Soares, F. M. and Souza, A. M. (2017). ***Neural network programming with Java*.** Packt Publishing Ltd. 86

Sommerville, I. (2011). **Software engineering 9th edition.** *ISBN-10*, 137035152. 51

Sun, X.-H. and Gustafson, J. L. (1991). **Toward a better parallel performance metric.** *Parallel Computing*, 17(10-11):1093–1109. 92

Van Der Aalst, W. M. and Ter Hofstede, A. H. (2005). **Yawl: yet another workflow language.** *Information systems*, 30(4):245–275. 38

Venkataraman, S., Yang, Z., Liu, D., Liang, E., Falaki, H., Meng, X., Xin, R., Ghodsi, A., Franklin, M., Stoica, I., and Zaharia, M. (2016). **SparkR: Scaling R programs with spark.** In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26-June-2016, pages 1099–1104. 29

Wainer, J. et al. (2007). **Métodos de pesquisa quantitativa e qualitativa para a ciência da computação.** *SBC - Atualização em informática*, 1:221–262. 96, 107

Wang, J., Crawl, D., and Altintas, I. (2009). **Kepler+ hadoop: a general architecture facilitating data-intensive applications in scientific workflow systems.** In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, page 12. ACM. 61

Xin, R. S., Gonzalez, J. E., Franklin, M. J., and Stoica, I. (2013). **Graphx: A resilient distributed graph system on spark.** In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM. 29

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). **Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing.** In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association. 28

Zaharia, M., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., and Venkataraman, S. (2016). **Apache spark: A unified engine for big data processing.** *Communications of the ACM*, 59(11):56–65. 19, 28

Zheng, Y., Xu, L., Wang, W., Zhou, W., and Ding, Y. (2017). **A reliability benchmark for big data systems on jointcloud.** In *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*, pages 306–310. IEEE. 28

Apêndice A Regras de transformação da Algebra Relacional

1. Cascata de Seleção - Uma condição de seleção conjuntiva pode ser desmembrada em uma sequência de operações de seleção (σ) individuais em cascata e vice-versa (Regra 10).

$$\sigma_{c1 \bigwedge c2}(R) \equiv \sigma_{c1}(\sigma_{c2}(R)) \tag{10}$$

2. Comutatividade de Seleção - As operações de seleção (σ) sejam elas conjuntivas ou disjuntivas, são comutativas (Regra 11).

$$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R)) \tag{11}$$

3. Cascata de Projeção - Em uma sequência de operações de projeção, basta considerar a última operação a ser executada. Todas as outras podem ser ignoradas. (Regra 12).

$$\pi_{c1}(\pi_{c2}(\pi_{c3}(R))) \equiv \pi_{c1}(R)) \tag{12}$$

4. Comutação de Seleção e Projeção - Se a condição de seleção c envolve apenas os atributos $A_1, A_2, ..., A_n$ da lista de projeção então as duas operações podem ser comutadas (Regra 13).

$$\pi_{A_1,A_2,...,A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1,A_2,...,A_n}(R)) \tag{13}$$

5. Comutatividade da Junção e Produto Cartesiano - Tanto a Junção quanto o Produto Cartesiano são comutativos (Regras 14 e 15). A ordem dos atributos não é a mesma na relação resultante mas isso não muda a semântica da consulta do ponto de vista da álgebra relacional.

$$R \bowtie_c S \equiv S \bowtie_c R \tag{14}$$

$$R \times S \equiv S \times R \tag{15}$$

6. Comutação da Seleção com a Junção (ou Produto Cartesiano) - A Seleção pode comutar tanto com a Junção quanto com o Produto Cartesiano, desde que a condição de seleção c envolva apenas os atributos de uma das relações participantes na operação de Junção (ou Produto Cartesiano), digamos R. (Regras 16 e 17).

$$\sigma_c(R \bowtie S) \equiv (\sigma_c(R)) \bowtie S \tag{16}$$

$$\sigma_c(R \times S) \equiv (\sigma_c(R)) \times S \tag{17}$$

No caso mais geral onde a condição c puder ser escrita como uma expressão conjuntiva na forma $c_1 \bigwedge c_2$ onde c_1 envolva apenas atributo de R e c_2 envolva apenas atributo de S, as operações podem ser comutadas como descrito nas regras 18 e 19

$$\sigma_c(R \bowtie S) \equiv (\sigma_{c_1}(R)) \bowtie (\sigma_{c_2}(S)) \tag{18}$$

$$\sigma_c(R \times S) \equiv (\sigma_{c_1}(R)) \times (\sigma_{c_2}(S)) \tag{19}$$

7. Comutação da Projeção com a Junção (ou Produto Cartesiano) - A Projeção pode comutar tanto com a Junção quanto com o Produto Cartesiano, desde que a lista de projeção seja $\{A_1, ..., A_n, B_1, ..., B_m\}$, onde $\{A_1, ..., A_n\}$ sejam atributos de R e $\{B_1, ..., B_m\}$ sejam atributos de S. É necessário também que a condição de junção envolva apenas atributos em L. Temos então neste caso as regras 20 e 21.

$$\pi_L(R \bowtie_c S) \equiv (\pi_{A_1,...,A_n}(R)) \bowtie (\pi_{B_1,...,B_m}(S)) \tag{20}$$

$$\pi_{_L}(R \times_{_c} S) \equiv (\pi_{A_1,...,A_n}(R)) \times (\pi_{B_1,...,B_m}(S)) \quad (21)$$

8. Comutatividade das operações de União e Intersecção - Tanto a União quanto a Intersecção são operações comutativas (Regras 22 e 23).

$$R \cup S \equiv S \cup R \quad (22)$$

$$R \cap S \equiv S \cap R \quad (23)$$

9. Associatividade das operações de União, Intersecção, Junção e Produto Cartesiano - Todas estas operações são associativas independentemente. (Regras 24, 25, 26 e 27).

$$(R \cup S) \cup T \equiv R \cup (S \cup T) \quad (24)$$

$$(R \cap S) \cap T \equiv R \cap (S \cap T) \quad (25)$$

$$(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T) \quad (26)$$

$$(R \times S) \times T \equiv R \times (S \times T) \quad (27)$$

10. Distributividade da Seleção - Distributividade da Seleção em relação as operações de União, Intersecção e Diferença. (Regras 28, 29 e 30).

$$\sigma_c(R \cup S) \equiv (\sigma_c(R)) \cup (\sigma_c(S)) \quad (28)$$

$$\sigma_c(R \cap S) \equiv (\sigma_c(R)) \cap (\sigma_c(S)) \quad (29)$$

$$\sigma_c(R - S) \equiv (\sigma_c(R)) - (\sigma_c(S)) \quad (30)$$

11. **Distributividade da Projeção em relação a União** - A Projeção pode distribuir sobre a União como pode ser visto na regra 31

$$\pi_{_L}(R \cup S) \equiv (\pi_{_L}(R)) \cup (\pi_{_L}(S)) \tag{31}$$

12. **Conversão de Seleção sobre Produto Cartesiano em Junção** - Quando temos uma operação de seleção de condição c sobre um Produto Cartesiano podemos transforma-la em uma Junção em c tal como mostra a regra 32

$$\sigma_c(R \times S) \equiv (R \bowtie_c S) \tag{32}$$

Apêndice B Prova de conceito - MDA

Para avaliar a independência de arquitetura e armazenamento do dado (*datasource*) fornecida pelo WfF, o *dataflow* `W2` (em duas versões `W2A` e `W2B`) foi submetido ao *Spark*. A diferença entre as versões `A` e `B` é apenas na parte relativa a execução (elemento `link` que liga a `relation` ao `resource`), de forma a gravar em tabelas relacionais de um SGBD Postgres ou arquivos Parquet via NFS. As credenciais para autenticação no Servidor Postgres e os outros dados de conexão (IP, porta, nome do banco de dados) foram fornecidos via variáveis de ambiente desacoplando do *workflow* em si [1].

Com isso foi possível comparar o desempenho da gravação entre os dois formatos e avaliar a independência da plataforma e de fonte de dado (*datasource*) usando a abstração criada na eDSL do *framework* WfF, baseado na MDA. O que o WfF favorece é a modificação das condições de materialização de forma simples, bastando modificar o elemento `link`. Na eDSL do WfF, a mudança de `W2A` para `W2B` se dá pela escolha do elemento `link` no elemento `job`, pois o `link` escolhido deve ligar a relação de saída ao recurso Postgres ou ao recurso Parquet.

Nas medidas de *speed-up* observou-se um comportamento assintótico a medida que aumenta-se o número de cores utilizados [2], com a curva de *speed-up* saturando em 11 cores quando a saída foi configurada para o formato Parquet. Em contrapartida, na gravação da saída em Postgres a curva de *speed-up* saturou em 4 cores. Na Seção 4.9, propõe-se um método para ajustar o número máximo de nodes evitando o desperdício de recursos. O método é particularmente útil em ambiente multiusuário de execução de *dataflows*. Foi observada uma eficiência medida de 53% menor do Postgres em relação ao Parquet. Das medidas de tempo decorrido, observou-se que a gravação do mesmo *dataset* no Postgres consome cinco vezes mais tempo do que a gravação no formato Parquet.

A avaliação preliminar verificou a transparência provida pelo WfF foi bem sucedida

[1] O WfF implementa um serviço de nomes simples análogo a um *domain name system* (DNS) para funcionar como *ServiceLocator*

[2] Este comportamento assintótico é o esperado

na prova de conceito. Também foi realizada uma avaliação quantitativa para investigar o comportamento da gravação em banco de dados Postgres comparado a armazenamento colunar do Parquet. Observou-se também que o *Spark* apresenta comportamento mais eficiente no *cluster* para *datasets* maiores.