UNIVERSIDADE FEDERAL DO PARANÁ

LUIZ CARLOS CAMARGO

DC4MT: UMA ABORDAGEM ORIENTADA A DADOS PARA TRANSFORMAÇÃO DE MODELOS

CURITIBA PR

2020

LUIZ CARLOS CAMARGO

DC4MT: UMA ABORDAGEM ORIENTADA A DADOS PARA TRANSFORMAÇÃO DE MODELOS

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Marcos Didonet Del Fabro.

CURITIBA PR

2020

Catalogação na Fonte: Sistema de Bibliotecas, UFPR
Biblioteca de Ciência e Tecnologia

C172d Camargo, Luiz Carlos
Dc4MT: uma abordagem orientada a dados para transformação de modelos [recurso eletrônico] / Luiz Carlos Camargo. – Curitiba, 2020.

Tese - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2020.

Orientador: Marcos Didonet Del Fabro

1. Engenharia de software. 2. Arquitetura de software baseada em modelo. 3. Software de computador (Desenvolvimento). I. Universidade Federal do Paraná. II. Del Fabro, Marcos Didonet. III. Título.

CDD: 005.1

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

MINISTÉRIO DA EDUCAÇÃO
SETOR DE CIENCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA - 40001016034P5

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **LUIZ CARLOS CAMARGO** intitulada: **DC4MT: uma Abordagem Orientada a Dados para Transformação de Modelos**, sob orientação do Prof. Dr. MARCOS DIDONET DEL FABRO, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.
A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 25 de Agosto de 2020.

Assinatura Eletrônica
25/08/2020 14:25:35.0
MARCOS DIDONET DEL FABRO
Presidente da Banca Examinadora

Assinatura Eletrônica
27/08/2020 13:37:44.0
ANDREY RICARDO PIMENTEL
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
25/08/2020 14:44:09.0
ERNANI GOTTARDO
Avaliador Externo (INSTITUTO FEDERAL DO RIO GRANDE DO SUL)

Assinatura Eletrônica
25/08/2020 16:36:47.0
FABIO PAULO BASSO
Avaliador Externo (UNIVERSIDADE FEDERAL DO PAMPA)

Rua Cel. Francisco H. dos Santos, 100 - Centro Politécnico da UFPR - CURITIBA - Paraná - Brasil
CEP 81531-980 - Tel: (41) 3361-3101 - E-mail: ppginf@inf.ufpr.br
Documento assinado eletronicamente de acordo com o disposto na legislação federal Decreto 8539 de 08 de outubro de 2015.
Gerado e autenticado pelo SIGA-UFPR, com a seguinte identificação única: 50668
Para autenticar este documento/assinatura, acesse https://www.prppg.ufpr.br/siga/visitante/autenticacaoassinaturas.jsp e insira o codigo 50668

À minha mãe Maria, à minha esposa Sidneia, à minha filha Jordana e aos meus filhos Lucas e Tiago.

AGRADECIMENTOS

A concretização de um Doutorado em tempo parcial, no meu caso o tempo foi compartilhado entre as atividades de Professor em uma instituição privada, requer determinação e resiliência. Ao longo da realização deste trabalho foram tantas as pessoas que de alguma forma ou em algum momento me incentivaram a seguir com a jornada, que agora está terminando. Não tenho como nominar todas essas pessoas aqui, mas deixo a elas meu agradecimento, Obrigado!

Expresso a minha gratidão ao Professor Marcos Didonet, não só pelo fato dele ter sido o orientador deste trabalho, mas pela maneira empática de como conduziu o processo de orientação. Didonet (como é conhecido no PPGInf) me concedeu a oportunidade de realizar este trabalho, me permitiu escolher o tema, soube apontar as mudanças. A convivência e as interações de orientação foram de uma aprendizagem valorosa, Muito Obrigado Professor!

Agradeço a minha família que sempre esteve ao meu lado, é claro que as vezes a minha esposa reclamava e dizia "você não sai desse computador", mas ela sabia que no desfecho, o computador era a principal ferramenta de trabalho. Meu agradecimento especial ao meu filho Lucas que sempre esteve à disposição para debater assuntos relacionados à tecnologia, essas discussões me faziam refletir a respeito desses assuntos, os quais fizeram parte das incertezas que o percurso de um Doutorado acarreta.

Meu agradecimento à Professora Edicarsia, que todo semestre fazia a distribuição da carga horária de trabalho (aulas e extensão), buscando minimizar o impacto dessas atividades ao andamento do Doutorado.

Agradeço aos membros da banca, Professores Andrey Ricardo Pimentel, Ernani Gottardo e Fábio Paulo Basso pelas críticas construtivas e também pelas sugestões.

Meu obrigado aos colegas de laboratório (FAES) por compartilhar o espaço de pesquisa e pelas discussões e descontrações que são essenciais ao processo de pesquisa. Ao PPGInf, os meus agradecimentos aos coordenadores do programa, aos professores e aos servidores da secretaria do PPGInf, sempre solícitos.

RESUMO

Transformações de Modelos são operações que recebem um conjunto de modelos como entrada e produzem um conjunto de modelos como saída, seguindo uma especificação. Há uma coleção diversificada de abordagens e ferramentas utilizadas para a especificação de diferentes tipos de transformações de modelos. A maioria dessas abordagens adota como estratégia a execução local e sequencial. No entanto, essas abordagens não estão totalmente aptas para processar modelos com grandes quantidades de elementos. VLMs (*Very Large Models*) são modelos que possuem milhões de elementos. Esses modelos estão presentes em domínios de aplicações como na indústria automotiva, modernização de sistemas legados, internet das coisas, redes sociais, entre outros domínios. Essas abordagens possuem lacunas para suportar o processamento desses VLMs. Por exemplo, para possibilitar a execução das transformações de modelos, considerando a escala do problema ou para melhoria de desempenho. Nesta tese é proposta a Dc4MT, uma abordagem para suportar transformações de VLMs com a aplicação e adaptação de técnicas relacionadas à distribuição de dados. A Dc4MT é uma abordagem Orientada a Dados (Dc - *Data-centric*) para ser aplicada no domínio da Engenharia Dirigida por Modelos (MDE - *Model Driven Engineering*). A abordagem é especificada, utilizando um framework de processamento distribuído, e define um conjunto de operações para a fragmentação, extração e transformação de modelos. A fragmentação é uma operação que divide os modelos de entrada (em formatos XMI ou JSON) em fragmentos, de modo que esses fragmentos possam ser distribuídos e processados de maneira paralela/distribuída. A extração é uma operação que processa os fragmentos do modelo de entrada e os traduz em um grafo acíclico, atribuindo um novo domínio de modelagem a esses fragmentos. A transformação de modelos na abordagem Dc4MT é uma operação que transforma modelos de entrada em modelos de saída (M2M) a partir do resultado da extração. As execuções de transformação podem ser em modo paralelo ou distribuído, com ou sem a intervenção no método de particionamento do framework disponível para melhorar o desempenho. Um conjunto de modelos de entrada (*datasets*) e os ambientes local (transformações paralelas) e distribuído (transformações distribuídas) são utilizados nos experimentos para validar a abordagem Dc4MT, sob os aspectos de factibilidade, desempenho e de escalabilidade. Os resultados desses experimentos, mostram que as operações de fragmentação e extração de modelos favorecem a transformação escalável de VLMs, reconstruindo a estrutura dos fragmentos em um grafo. A operação de extração é executada em modo paralelo/distribuído. Além disso, os aspectos como a imutabilidade, lazy-evaluate e o paralelismo implícito presentes na Dc4MT, permitem o processamento paralelo/distribuído de regras de transformação em uma plataforma escalável.

Palavras-chave: Abordagem Orientada a Dados. Engenharia Dirigida por Modelos. Transformação Paralela de Modelos. Transformação Distribuída de Modelos.

ABSTRACT

Model Transformations are operations that receive a set of source models as input and produce a set of target models as output, following a specification. There is a variety of approaches and tools used for the specification of different types of model transformation. Most of these approaches adopt for model transformation the local and sequential execution strategy. However, these approaches not fully adapted for processing models with large amounts of elements. VLMs (*Very Large Models*) are models with millions of elements. These models are present in application domains such as the automotive industry, modernization of legacy systems, internet of things, social networks, among others. These approaches have gaps to support the processing of these increasingly larger models. For example, to enable model transformations, considering the scale of the problem or to improve performance. In this thesis, the Dc4MT is proposed such as an approach to support transformation of VLMs, applying and adapting distribution techniques of data. The Dc4MT is a Data-centric (Dc) approach for applying in Model Driven Engineering (MDE). The approach will be specified using a distributed processing framework, and defines a set of operations for fragmentation, extraction, and transformation of models. The fragmentation is an operation that splits the input models (in the XMI or JSON formats) in a way that the fragments can be processed in parallel/distributed. The extraction is an operation that processes the fragments of the input model in parallel and translates them to an acyclic graph, assigning a new modeling domain to these fragments. The model transformation in Dc4MT is an operation that transforms input models in output models (M2M) from the results of the extraction. The transformation executions can be parallel or distributed with or without the intervention in the framework partitioning method to improve the performance. A set of input models (datasets) and the local (parallel transformations) and distributed (distributed transformations) environments are used in the experiments to validate the Dc4MT approach, in terms of feasibility, performance, and scalability. The results of the experiments show that the model fragmentation and extraction operations favor the scalable transformation of models, reconstructing the structure of the fragments in a graph. The extraction operation is executed on parallel/distributed way. Moreover, aspects such as immutability, lazy-evaluation, and implicit parallelism present in Dc4MT, allowing the parallel/distributed processing of transformation rules on a scalable platform.

Keywords: Data-centric Approach. Model Driven Engineering. Parallel Model Transformation. Distributed Model Transformation.

LISTA DE FIGURAS

1.1 Organização de Capítulos 19

2.1 MDE em Níveis de Abstração, adaptada de (Brambilla et al., 2017) 21
2.2 Um Esquema Representativo de Transformação de Modelos 22
2.3 Metamodelos Families e Persons 24
2.4 Uma Visão Geral da Computação nos Frameworks MapReduce e Spark . . . 29
2.5 Arquitetura do Spark em Modo Cluster com Três Nós 30
2.6 Um Exemplo de DataFrame 31
2.7 Família March em um GraphFrames 32

4.1 Uma Visão Geral de uma Implementação Modular da Dc4MT 58
4.2 Passos das Principais Atividades das Operações Dc4MT 59
4.3 Um Exemplo do Fluxo do Modelo de Comunicação 60
4.4 Tipos de Dependência entre Partições RDDs 62
4.5 Fluxo da Operação de Fragmentação de Modelos 64
4.6 Modelo para Tradução de Modelos em Grafos 66
4.7 Tradução de Elementos do Modelo Families para um GraphFrames 70
4.8 Modelos Families e Class Visualizados em Formato Grafo 71

5.1 Resultado das Fragmentações de Modelos 77
5.2 Transformações dos Modelos `Families` e `Class` 80
5.3 Transformações dos Modelos `Imdb` e `DBLP` 83
5.4 Resultado das Transformações Distribuídas de `Families2Persons` 87
5.5 Resultado das Transformações Distribuídas de `Class2Relational` 88
5.6 Transformações Distribuídas dos Modelos `Imdb` 88

LISTA DE TABELAS

2.1 Classificação de Transformação de Modelos 23

3.1 Classificação de Trabalhos Relacionados à TP/DM com ou sem PM (parte 1) 38

3.2 Classificação de Trabalhos Relacionados à TP/DM com ou sem PM (parte 2) 47

3.3 Classificação de Trabalhos Relacionados à DC 55

3.4 Comparação de Aspectos da Dc4MT com um Subconjunto dos Trabalhos Relacionados 57

5.1 Tipos de Dataset Utilizados nos Experimentos 76

5.2 Extração dos Modelos Família para GraphFrames 78

5.3 Resultado da Extração dos Modelos Class para o GraphFrames 78

5.4 Extração dos Datasets IMDB para GraphFrames 78

5.5 Extração dos Modelos Families em XMI e JSON 79

5.6 Resultado das Transformações Paralelas - Families2Persons. 81

5.7 Transformações Class2Relational. 82

5.8 Impacto do Shuffle Partitions nas Execuções Paralelas das Transformações . 82

5.9 Transformações de Modelos Imdb 84

5.10 Transformações do Modelo DBLP 84

5.11 Transformações Distribuídas Families2Persons 87

5.12 Transformações Distribuídas Class2Relational 87

5.13 Transformação Distribuída Imdb 89

5.14 Transformações Distribuídas DBLP 89

5.15 Impacto do *Shuffle Partition* nas Execuções Distribuídas das Transformações 89

LISTA DE ACRÔNIMOS

ACID	Atomicidade, Consistência, Isolamento e Durabilidade
API	*Application Programming Interface*
ATL	*ATLAS Transformation Language*
DAG	*Directed Acyclic Graph*
Dc	*Data-centric*
DINF	Departamento de Informática
DSL	*Domain-specific Language*
EMF	*Eclipse Modeling Framework*
ETL	*Epsilon Transformation Language*
IDE	*Integrated Development Environment*
JSON	*JavaScript Object Notation*
MDE	*Model Driven Engineering*
MOF	Serviço de MetaObjeto (*MetaObject Facility*)
MT	*Model Transformations*
NoSQL	Not Only Structured Query Language
OCL	*Object Constraint Language*
OMG	*Object Management Group*
PIM	*Platform Independent Model*
PSM	*Platform Specific Modelo*
PPGINF	Programa de Pós-Graduação em Informática
QVT	*Query View Transformation*
RDD	*Resilient Distributed Dataset*
SE	*Software Engineering*
TM	Transformação de Modelos
TP/DM	Transformação Paralela e/ou Distribuída de Modelos
UFPR	Universidade Federal do Paraná
UML	*Unified Modelling Language*
VLM	*Very Large Model*
XMI	*XML Metadata Interchange*
XML	*eXtensible Markup Language*

LISTA DE SÍMBOLOS

At_1	Denota o Atributo 1
At_2	Denota o Atributo 2
A_x	Denota po Ambiente de Execução
Cl_1	Denota a Classe 1
CL_2	Denota a Classe 2
Dt_1	Denota o Tipo de dados 1
E	Denota Arestas
E_M	Indica Elementos de Modelo
F	Fragmentos
F_E	Indica Fragmentos de Entrada
F_N	Indica Quantidade de Fragmentos
G	Denota Grafo
GF	Indica GraphFrames
M	Denota Modelo
M_e	Indica Modelo de Entrada
N_M	Denota Nó Master
N_{no}	Denota o número de nós de uma máquina ou de um *cluster* de máquinas
N_W	Denota Nó Worker
Pc_1	Denota o Pacote 1
S_G	Denota um Subgrafo
V	Denota Vértices
$\bowtie$	Denota Operação de Junção
$\emptyset$	Denota Conjunto vazio

SUMÁRIO

1 INTRODUÇÃO . . . 13
1.1 JUSTIFICATIVAS E MOTIVAÇÕES . . . 15
1.2 OBJETIVOS . . . 17
1.3 CONTRIBUIÇÕES . . . 18
1.4 ORGANIZAÇÃO DA TESE . . . 19
2 FUNDAMENTAÇÃO TEÓRICA . . . 20
2.1 ENGENHARIA DIRIGIDA POR MODELOS . . . 20
2.1.1 Transformação de Modelos . . . 21
2.1.2 Linguagem de Transformação ATL . . . 24
2.2 ABORDAGEM ORIENTADA A DADOS . . . 27
2.2.1 Framework Spark . . . 28
2.3 PARTICIONAMENTO DE MODELOS. . . . 32
2.3.1 Anotações. . . . 32
2.3.2 Particionamento de Grafos . . . 33
2.3.3 Técnica *Slice* . . . 34
2.4 CONSIDERAÇÕES FINAIS . . . 35
3 TRABALHOS RELACIONADOS . . . 36
3.1 TRANSFORMAÇÃO PARALELA E/OU DISTRIBUíDA DE MODELOS . 36
3.1.1 TP/DM com Particionamento ou Fragmentação do Modelo de Entrada . . . 38
3.1.2 TP/DM sem Particionamento ou Fragmentação do Modelo de Entrada . . . 39
3.1.3 Particionamento/Fragmentação de Modelos. . . . 42
3.2 TRABALHOS RELACIONADOS À ABORDAGEM ORIENTADA A DADOS . . . 48
3.3 CONSIDERAÇÕES FINAIS . . . 56
4 A ABORDAGEM DC4MT. . . . 58
4.1 VISÃO GERAL DA ABORDAGEM DC4MT . . . 58
4.2 MODELO DE COMUNICAÇÃO DO AMBIENTE DE EXECUÇÃO . . . 60
4.3 FRAGMENTAÇÃO DE MODELOS . . . 63
4.4 EXTRAÇÃO DE MODELOS . . . 65
4.4.1 Resolução de Referências . . . 68
4.4.2 Visualização de Densidade de Referências. . . . 70
4.5 ESPECIFICAÇÃO DE TRANSFORMAÇÕES DE MODELOS . . . 71
4.6 CONSIDERAÇÕES FINAIS . . . 74

5 EXPERIMENTOS . . . 75
5.1 EXTRAÇÃO PARALELA DE MODELOS . . . 77
5.2 TRANSFORMAÇÕES PARALELA DE MODELOS . . . 79
5.2.1 Considerações sobre os resultados - Execuções paralela em modo local . . . 85
5.3 TRANSFORMAÇÕES DISTRIBUíDAS DE MODELOS . . . 85
5.3.1 Considerações sobre os resultados - Execuções Distribuídas . . . 90
5.4 CONSIDERAÇÕES FINAIS . . . 91
6 CONCLUSÃO . . . 92
6.1 TRABALHOS FUTUROS . . . 93
REFERÊNCIAS . . . 95
APÊNDICE A – REGRAS DE TRANSFORMAÇÃO DE MODELOS . . . 107

1 INTRODUÇÃO

Modelos são essenciais em várias áreas de conhecimento, são úteis para a abstração da realidade ou parte dela. Um modelo pode ser considerado como uma redução das características do objeto modelado ou o mapeamento generalizado desse objeto (Brambilla et al., 2017; Mens e Van Gorp, 2006). Modelos servem para representar conceitos, sistemas ou processos (Brambilla et al., 2017). Modelos são usados sob diferentes perspectivas. Por exemplo, em sistemas, um modelo pode expressar uma visão comportamental ou estrutural de um sistema ou de um segmento dele. Um sistema é considerado como um conjunto de entidades inter-relacionadas que interagem no desempenho de uma função (Klir, 2013). Conforme os modelos são criados, sistematicamente eles podem ser transformados (e.g., transformar um diagrama de classes em tabelas de um banco de dados relacional).

A Engenharia Dirigida por Modelos (MDE *Model Driven Engineering*) é uma abordagem da Engenharia de Software (ES), em que o modelo é o foco principal. A MDE provê um conjunto de técnicas para criar, manter e transformar modelos, desempenhando um importante papel no processo de desenvolvimento de software (Brambilla et al., 2017; Kent, 2002; Schmidt, 2006). Novos sistemas surgem, cuja complexidade e o tamanho têm aumentados, exigindo mais processamento computacional em ambientes paralelos/distribuídos (Zomaya e Sakr, 2017; Qin et al., 2016). Os modelos que representam esses sistemas exigem mais capacidade computacional e eficiência para a criação e manipulação de seus elementos.

As Transformações de Modelos (TMs) são consideradas artefatos essenciais para a implementação de operações entre modelos, possibilitando a manipulação de modelos (Brambilla et al., 2017; Stahl et al., 2006; Kleiner et al., 2013; Fabro, 2007; Burgueno, 2013). Uma TM pode ser unidirecional ou bidirecional. No modo unidirecional, modelos de entrada (origem) são transformados em modelos de saída (destino). Enquanto que no modo bidirecional, a transformação é realizada da origem para o destino e vice-versa. Além disso, a TM deve fornecer um conjunto de tarefas que favoreça o processamento de modelos, metamodelos e regras de transformação de maneira interdependente (Paige et al., 2016).

Há um variado conjunto de abordagens e linguagens com diferentes características, esse conjunto é utilizado em diversos tipos de transformação de modelos (Kahani et al., 2019): GrGen é utilizada para transformação de grafos (Jakumeit et al., 2010); UML-RSDS, uma ferramenta de MDE para propósito geral (Lano e Kolahdouz-Rahimi, 2017); Kermeta, uma linguagem imperativa (Drey et al., 2017); QVT-R, uma linguagem declarativa (OMG, 2016a); A Linguagem de Transformação Epsilon (ETL *Epsilon Transformation Language*) (Kolovos et al., 2008) e a Linguagem de Transformação ATL (ATL *ATLAS Transformation Language*), uma linguagem híbrida, é aquela que aceita os estilos declarativo e imperativo de programação. A ATL é de domínio específico (DSL *Domain Specific Language*), e é uma das soluções mais utilizadas nas especificações de TMs (Jouault e Kurtev, 2005; Eclipse, 2019a). Também há abordagens dedicadas à TMs baseada em Resolvedores (Macedo e Cunha, 2013; Kleiner et al., 2013; Macedo et al., 2013; Macedo e Cunha, 2016) e baseadas em Exemplos (Kessentini et al., 2010; Kappel et al., 2012; García-Magariño et al., 2009). Contudo, a maioria dessas abordagens adota como estratégia a execução local e sequencial para a transformação de modelos. Com tal estratégia, essas abordagens não possibilitam a execução paralela/distribuída de modelos,

condicionando o processamento de modelos com grandes quantidades de elementos (VLMs *Very Large Models*) à capacidade do ambiente de execução.

Um VLM é formado por milhões de elementos. VLMs estão presentes em domínios específicos como na indústria automotiva, engenharia civil e em Linha de Produtos de Software (SPL *Software Product Lines*) (Clements e Northrop, 2001), ou ainda na modernização de sistemas legados (Gómez et al., 2015). Além disso, surgem novas aplicações envolvendo domínios como a Internet das Coisas (IoT *Internet of Things*), repositórios de dados abertos (*OpenData*), redes sociais, entre outros que exigem computação intensiva para a manipulação de seus artefatos (Ahlgren et al., 2016). Trabalhos como os de Burgueño et al. (2016), Pagán et al. (2015), Benelallam et al. (2018), Daniel (2017), e Tisi et al. (2013) indicam que a MDE ainda não está apta o suficiente para tratar de domínios de aplicações que exigem escalabilidade na manipulação de modelos com grandes quantidades de elementos.

O processamento escalável de VLMs ainda é um desafio para a MDE e tem estimulado a realização de trabalhos, como os de Burgueño et al. (2016), Tisi et al. (2013), Benelallam et al. (2015), Daniel (2017). Esses trabalhos combinam reconhecidas linguagens de transformações de modelos, como a ATL, com *frameworks* de processamento paralelo e/ou distribuído. O resultado desses trabalhos mostram avanços no que diz respeito ao uso do paralelismo implícito (abstração da coordenação entre processos e dados em ambientes paralelos/distribuídos) e a escalabilidade no processamento de transformações de grandes modelos. Exemplos desses avanços são encontrados na integração de abordagens de transformação de modelos com o MapReduce (Dean e Ghemawat, 2008; Benelallam et al., 2018), com o framework Linda (Gelernter, 1985; Burgueño et al., 2016), e na transformação de modelos em paralelo com uma implementação *multi-thread* utilizando a ATL (Tisi et al., 2013). No entanto, as soluções de processamento paralelo em memória compartilhada, propostas por Tisi et al. (2013) e Burgueño et al. (2016), exigem que o modelo de entrada seja carregado por completo em memória principal. Tal estratégia pode comprometer a capacidade de processamento de VLMs. Para transmitir e embaralhar (*shuffing phase*) os elementos do modelo sob processamento, as soluções baseadas em MapReduce (Benelallam et al., 2018; Aracil e Ruiz, 2017) consomem tempo e memória durante a transformação de modelos nas fases Map e Reduce.

Frameworks que dispõem de paralelismo implícito permitem o reúso de construtores de comunicação, coordenação de processos e de tarefas. Tal reutilização possibilita o gerenciamento automático de dados e de tarefas, direcionando o desenvolvimento implícito de aplicações paralelas/distribuídas (Wilkinson e Allen, 1999). Mesmo adotando esses frameworks, a programação distribuída requer familiaridade com a concorrência e distribuição de processos, paralelismo, balanceamento de carga computacional e localidade de dados (Benelallam, 2016). Além disso, o processamento de modelos não é uma tarefa elementar, por exemplo, quando comparado ao processamento de dados organizados em estrutura simplificada e unidimensional. Modelos tendem à ter densas estruturas multidimensionais com interconexões entre elementos, podendo ser vistos como um grafo formado por vértices e conectados por arestas (Mezic et al., 2019; Giese et al., 2012).

Em diferentes contextos sustentados pela computação em nuvem e por soluções de *Big Data*, novos cenários para sistemas Orientados a Dados (Dc *Data-centric*) estão surgindo (Zomaya e Sakr, 2017). A maioria desses sistemas tem como foco o processamento de dados, independente da estrutura desses dados. As aplicações Dc utilizam requisitos como a programação de alto nível, estilo declarativo e a distribuição automática de tarefas e dados em ambientes de execução paralela/distribuída (Alvaro et al., 2010). Esses

requisitos são fornecidos por frameworks como o MapReduce, Hadoop (Apache, 2019c) e Spark (Apache, 2019d) em aplicações paralelas/distribuídas.

1.1 JUSTIFICATIVAS E MOTIVAÇÕES

A transformação de modelos em larga escala é um problema que ainda não foi totalmente solucionado pela MDE. Os mecanismos presentes nas principais abordagens de transformação de modelos como QVT, ATL, ETL, VIATRA, entre outras, não foram projetadas para ambientes paralelos/distribuídos (Kahani et al., 2019; Burgueño et al., 2016, 2015; Clasen et al., 2012). Tais abordagens adotam somente o XMI (*XML Metadata Interchange*) como formato padrão para os modelos de entrada, uma vez que há outros formatos para a serialização de dados como o YAML[1] e o JSON[2]. Tal limitação tem dificultado as iniciativas que buscam levar as soluções de MDE para computação em nuvem (Muzaffar et al., 2017), também denominadas de MDE como um Serviço (Basso et al., 2017). Além disso, essas abordagens exigem que o modelo de entrada esteja disponível na memória principal para o processamento da transformação (Tisi et al., 2013). A persistência escalável de VLMs é fornecida pelo framework NeoEMF, o qual admite modelos originados do Framework Eclipse de Modelagem (EMF), mas não inclui transformação de modelos (Daniel et al., 2016).

Clasen et al. (2012), discutem as implicações relacionadas a uma possível junção da MDE e a computação em nuvem, incluindo armazenamento distribuído de dados, transformações de modelos em modo paralelo e/ou distribuído utilizando o framework MapReduce. Essa discussão tem motivado o desenvolvimento de trabalhos inserindo a MDE em ambientes escaláveis e distribuídos, tais como:

- execuções paralela (Tisi et al., 2013) e incremental (Calvar et al., 2019) de transformações de modelos em ATL;
- utilização do JlinTra (uma combinação da linguagem Java com o Framework Linda) para execuções paralela de transformações de modelos (Burgueño, 2016);
- abordagem ATL-MR para transformação paralela/distribuída de modelos em que a ATL é acoplada ao framework MapReduce. Incluem nessa abordagem a persistência e o particionamento de modelos (Benelallam, 2016);
- transformação relacional e distribuída de modelos como uma extensão da abordagem ATL-MR (Benelallam et al., 2018);
- uma abordagem para o armazenamento em NoSQL de modelos em larga escala incluindo consulta e transformação escaláveis de modelos por meio do Framework Gremlin-ATL (junção das linguagens Gremlin[3] e ATL) (Daniel, 2017);
- CloudTL uma linguagem para o processamento de modelos no formato Ecore e em modo cliente-servidor, a qual é apoiada por camadas de aplicações WEB (Aracil e Ruiz, 2017);

[1]https://yaml.org/
[2]https://www.json.org
[3]https://tinkerpop.apache.org/gremlin.html

Tais abordagens processam regras de transformações de modelos de maneira escalável e estão integradas ao ecossistema do framework EMF (Eclipse, 2019c) e ao formato XMI. Contudo, essas abordagens não unificam em uma plataforma questões como fragmentação, extração e a transformação paralela/distribuída de modelos. Além disso, operações sobre modelos em XMI podem requerer a inserção por completo de modelos nesse formato na memória principal. Neste caso, a memória de uma simples máquina pode não ser suficiente para um VLM. Estratégias como *lazy-loading*, *streaming* ou carregamento parcial podem minimizar esse impasse (Daniel, 2017; Benelallam et al., 2016). Segundo Smolders (2017) e Hili (2016), a serialização de modelos expressado no formato JSON (*JavaScript Object Notation*) tem sido mais eficiente quando comparada com a serialização em XMI. Além disso, modelos de dados baseados em JSON são adotados em banco de dados NoSQL (*Not Only SQL*) como o Apache CouchDB [4], MongoDB [5], entre outros.

No entanto, nenhuma dessas abordagens adotam uma estratégia de fragmentação [6] do modelo de entrada de modo que os fragmentos possam ser carregados como *streaming* e processados em paralelo/distribuído. Não há nesses trabalhos, um mecanismo que permita a visualização da complexidade estrutural do modelo de entrada no que diz respeito à interconectividade entre seus elementos (referências entre elementos do modelo). O uso de estratégias de particionamento e paralelismo a fim de minimizar a troca de mensagens entre os processos durante a execução da transformação, o uso de estratégias de particionamento e o paralelismo ainda são pouco explorados. Além dessas questões, há as recentes discussões envolvendo uma análise da pesquisa, da prática e do estado da arte em MDE promovidas por Bucchiarone et al. (2020). Há também o framework MODA (*Models and Data*), proposto por Combemale et al. (2020) como um resultado da junção das abordagens MDE e Dc na integração de modelos heterogêneos e seus respectivos dados. Essas questões justificam e motivam a realização desta tese. Isto posto, a questão principal de pesquisa é apresentada:

> **Como prover uma abordagem escalável para transformação de modelos em uma plataforma, incluindo a fragmentação, extração e o particionamento de VLMs?**

Para balizar os diferentes aspectos contidos na questão principal, foram elaboradas três questões, as quais são tratadas ao longo desta tese:

RQ1 Como extrair elementos de VLMs em mais de um formato de maneira escalável?

RQ2 É possível particionar modelos visando o desempenho da execução da transformação de modelos?

RQ3 Como proporcionar uma plataforma unificada para o processamento paralelo/distribuído de modelos?

A resposta para cada questão está endereçada na seguinte hipótese: uma estratégia de fragmentação do modelo de entrada pode facilitar a extração e a distribuição dos fragmentos entre os nós de um ambiente paralelo/distribuído; inclui-se os requisitos da abordagem Dc, como a programação de alto nível, o paralelismo implícito, e a adoção de uma plataforma para o processamento escalável de modelos.

[4] http://couchdb.apache.org/

[5] https://www.mongodb.com/

[6] Nesta tese os termos fragmentação e fragmento são adotados para os modelos de entrada. Enquanto que o termo particionamento é usado na operação de transformação.

1.2 OBJETIVOS

O principal objetivo desta tese é propor uma abordagem orientada a dados para transformação paralela e distribuída de modelos. Inclui nessa abordagem a fragmentação, extração e o particionamento de modelos. A fragmentação de modelos busca dividir o modelo de entrada em fragmentos, os quais possam ser processados em modo paralelo/distribuído. A extração de modelos consiste em extrair elementos de um determinado espaço de domínio de modelagem para o um novo domínio, neste caso em grafo representado por GraphFrames. O particionamento e a distribuição de modelos são orquestrados pelo framework de paralelismo implícito, porém uma estratégia de particionamento é adotada como o controle da quantidade de partições de modelos em relação ao tamanho do modelo a ser processado. Para alcançar o objetivo geral, os seguintes objetivos secundários são definidos:

a) fragmentar modelos de entrada: uma estratégia de fragmentação consistente de modelos, de modo que os fragmentos possam ser distribuídos e processados em paralelo;

b) desenvolver um extrator de modelos: a extração de modelos busca traduzir os modelos de entrada para o espaço de domínio de modelagem da Dc4MT (grafo direcionado), já que esses modelos pertencem a um domínio de modelagem diferente;

c) estabelecer estratégias para o particionamento de modelos: a intervenção no particionamento do ambiente de paralelismo implícito pode favorecer o desempenho do processamento paralelo/distribuído das regras de transformação;

d) desenvolver um conjunto de regras de transformação para diferentes modelos: um modelo de transformação composto por filtros e regras para processar partições de modelos de maneira paralela e ou distribuída;

e) validar a abordagem: é indispensável a realização de experimentos para validação da escalabilidade da Dc4MT, usando VLMs e comparando com abordagens similares.

O desenvolvimento da abordagem Dc4MT está pautado na fragmentação, extração e transformação de modelos, no paralelismo implícito e particionamento de dados. Os dois últimos são fornecidos pelo próprio ambiente de processamento paralelo/distribuído (Spark) adotado para a Dc4MT. Além disso, aspectos como operações relacionais, imutabilidade e *lazy-evaluate* são utilizados no desenvolvimento da Dc4MT, cujas contribuições são apresentadas na próxima seção.

1.3 CONTRIBUIÇÕES

Nesta tese, incluem-se à abordagem Dc4MT as seguintes contribuições:

- uma alternativa para a fragmentação de modelos. Fragmentação balanceada de modelos, em que uma estratégia de fragmentação é adotada, permitindo a execução paralela/distribuída de modelos;
- a extração paralela e distribuída de modelos é uma alternativa de carregamento tardio de modelos, por demanda do ambiente de processamento, sem a necessidade do carregamento de uma vez para a memória principal;
- uma abordagem para a tradução de modelos de entrada em formatos XMI ou JSON para um formato de grafo, permitindo que esses modelos passem a pertencer a um único espaço técnico de modelagem;
- o ganho de desempenho no processamento de VLMs, por meio de um método que interfere no particionamento implícito de modelos é uma maneira de minimizar a dependência de dados entre os nós do ambiente durante a execução paralela/distribuída.

A visualização de modelos e a contabilização de vértices e arestas de modelos são úteis para a elaboração de estratégias de manipulação desses modelos, uma vez que se tem a dimensão da estrutura do modelo a ser processado. Além disso, aspectos como *lazy-load* e *lazy-evaluate*, paralelismo implícito e a especificação declarativa das regras de transformação são contribuições desta tese no âmbito de transformação de VLMs em uma plataforma escalável.

1.4 ORGANIZAÇÃO DA TESE

Na Figura 1.1, encontram-se a organização desta tese e os relacionamentos entre seus capítulos.

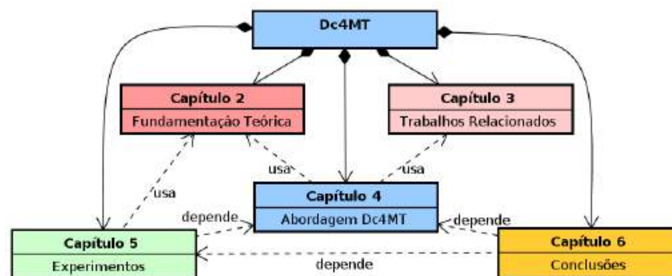


Figura 1.1: Organização de Capítulos

No Capítulo 2, são introduzidos os conceitos das abordagens MDE e Dc seguida pela apresentação de estratégias de Fragmentação e de Particionamento de Modelos. No Capítulo 3, têm-se uma descrição, classificação e análise detalhada dos trabalhos relacionados à transformação paralela/distribuída de modelos e de aplicações da abordagem Dc. No Capítulo 4, a abordagem Dc4MT é descrita. No Capitulo 5, os resultados dos experimentos em modo local e distribuído são apresentados e discutidos. A conclusão está no Capítulo 6.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 ENGENHARIA DIRIGIDA POR MODELOS

Modelos desempenham um importante papel em diferentes contextos, modelos propiciam o entendimento de questões complexas de um sistema para uma determinada finalidade (Wagner, 2014). Modelos podem ser usados para simplificar a realidade sob várias perspectivas, e ao mesmo tempo, são fundamentais para o entendimento de conceitos básicos sobre um campo específico de conhecimento. Na Engenharia de Software (ES), modelos são necessários para: representações abstratas de software, descrição de um domínio de problema, especificações de código fonte, composição da documentação de um sistema, entre outras representações. Segundo Stachowiak (1973), um modelo deve ter as seguintes características: *mapeamento*: um modelo pode ser considerado como uma representação do sistema original e que pode ser expressado por meio de uma linguagem de modelagem; *redução:* sequer todas as propriedades do sistema são mapeadas para o modelo, tal modelo é somente um subconjunto do sistema original; *pragmático:* um modelo precisa ser utilizável no lugar do original com relação a algum propósito ou objetivo.

A MDE (*Model Driven Engineering*) é uma abordagem da ES direcionada ao desenvolvimento de sistemas, em que os modelos são artefatos fundamentais à MDE. A MDE fornece mecanismos para lidar com os artefatos de software utilizados e produzidos durante o ciclo de vida de desenvolvimento de software, os quais podem ser modificados, atualizados ou processados para diferentes propósitos (Brambilla et al., 2017; Babau et al., 2010; Fabro, 2007). Além do ciclo-de-vida de desenvolvimento de software, a MDE é aplicada em outros domínios. Por exemplo, na modernização de software, em que os códigos fontes de sistemas legados são utilizados para criar modelos abstratos. Por meio desses modelos, arquiteturas legadas de software podem ser avaliadas e refatoradas. Além disso, pode-se gerar novos códigos para uma plataforma específica a partir de tais modelos. (Benelallam, 2016; Daniel, 2017).

Existem diferentes tipos de modelos direcionados ao desenvolvimento de software, e por meio desses modelos a MDE fornece uma visão integrada de várias propriedades e dimensões do sistema de software. Essa visão é organizada em três níveis de abstração (`M3`, `M2` e `M1`), nos quais Metametamodelo, Metamodelo e Modelo são organizados (Wagner, 2014; Brambilla et al., 2017; da Silva, 2015):

- Metametamodelo é um modelo que descreve um metamodelo de maneira autocontida para um dado contexto. Também é chamado de nível de metametamodelo auto-reflexivo superior e descreve uma representação para os Modelos e Metamodelos de um determinado domínio;

- Metamodelo define conceitos, relacionamentos e a semântica para que o(a) desenvolvedor(a) possa criar modelos em diferentes ferramentas de modelagem. Um metamodelo descreve outro modelo, definindo os tipos de elementos e o relacionamento entre eles;

- Modelo é uma abstração do mundo real formado por um conjunto de elementos, os quais podem estar relacionados entre si. Um modelo é uma instância do metamodelo.

Na Figura 2.1, são representados os níveis de abstração: `M3`, `M2` e `M1` com as respectivas relações de conformidade indicada pela relação `conforme` (`conformsTo` ou `c2`). As instâncias de metametamodelo e metamodelo são assinaladas pela relação de dependência `instância`. O Metametamodelo é representado no nível `M3` de abstração da MDE. Para expressar Metametamodelos, o OMG (*Object Management Group*) propôs a MOF (*MetaObject Facility*) (OMG, 2016b) uma arquitetura padrão para a metamodelagem, tendo a OCL (*Object Constraint Language*) (OMG, 2014) como uma linguagem para especificar restrições em modelos baseados em MOF. Além da MOF, Jouault e Bézivin (2006) propôs a KM3 (*Kernel MetaMetaModel*), uma DSL para especificar Metamodelos, cujas restrições para os Metamodelos são expressadas por meio de fórmulas.

Em conformidade (`conforme`) com o Metametamodelo, o Metamodelo é representado no nível `M2` de abstração, no qual são descritos os conceitos usados na camada `M1` para definir os elementos do modelo e os seus relacionamentos. Um Metamodelo é uma instância (`instância`) do Metametamodelo e pode ser expressado em diferentes tecnologias como MOF, UML (*Unified Modeling Language*)[1] e o EMF (*Eclipse Modeling Framework*)[2].

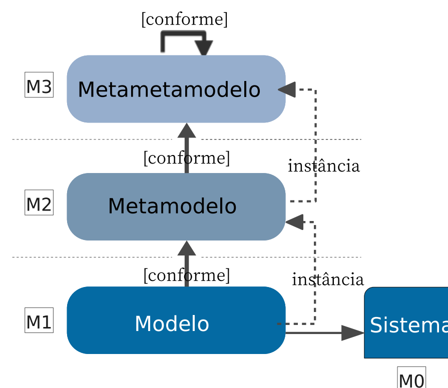


Figura 2.1: MDE em Níveis de Abstração, adaptada de (Brambilla et al., 2017)

Um modelo é representado no nível `M1` de abstração e deve estar em conformidade com um metamodelo, descrevendo os conceitos de um sistema. Todos os modelos são instâncias (`instancia`) de um Metamodelo. Modelos são criados para diferentes domínios de aplicações e em diferentes níveis de abstração como nível de negócio, requisitos técnicos ou nível de projeto de software. Modelos podem ser expressados em uma linguagem de modelagem (e.g., UML) e capturar diferentes visões de um sistema, como estrutural (e.g., diagrama de classes) e comportamental (e.g., diagrama de sequência) (da Silva, 2015). Por fim, o nível `M0` que inclui os elementos de um sistema definidos no Modelo (M1), sejam eles de um ambiente computacional ou de um domínio de aplicação. Conforme os níveis de abstração apresentados na Figura 2.1, modelos podem ser criados e sistematicamente transformados em modelos concretos contendo elementos de software.

2.1.1 Transformação de Modelos

Uma TM consiste em receber um modelo como entrada e transformá-lo em um ou mais modelos de saída, conforme a especificação dessa transformação (Tehrani et al., 2016). Há outros conceitos sobre TM: de acordo com Brambilla et al. (2017), a TM é um

[1]https://www.omg.org/spec/UML/
[2]https://www.eclipse.org/modeling/emf/

componente fundamental para a MDE, o qual permite o mapeamento entre diferentes modelos; Macedo (2014) considera a TM como o âmago da MDE; TMs são fatores-chave para a realização de operações entre diferentes modelos (Kleiner et al., 2013); TM é uma operação que recebe um conjunto de modelos como entrada, visita os elementos desses modelos e produz um conjunto de modelos como saída (Fabro, 2007); uma transformação de modelos define a tradução de um modelo em outro, ambos comum a uma especificação escrita em uma linguagem específica (Berramla et al., 2016).

A transformação de modelos é uma operação que consiste na produção automática de um ou mais modelos de saída, a partir de modelos de entrada e de acordo com a especificação da transformação (Daniel, 2017). Na Figura 2.2, é ilustrado um esquema para complementar os conceitos apresentados e expressar uma visão geral sobre TM. De acordo com esse esquema, a relação (`conforme`) estabelece uma relação de conformidade entre o Metametamodelo e os Metamodelos de Entrada e Saída e aos demais componentes do esquema. A relação `usa`, representa as ligações (*links*) entre os elementos dos Modelos em operação de transformação por meio dos Metamodelos. O esquema também representa o aspecto de bidirecionalidade, em que o Modelo de Entrada pode ser uma entrada ou saída (`entrada/saída`) em um processo de transformação bidirecional, da mesma forma para o Modelo de Saída.

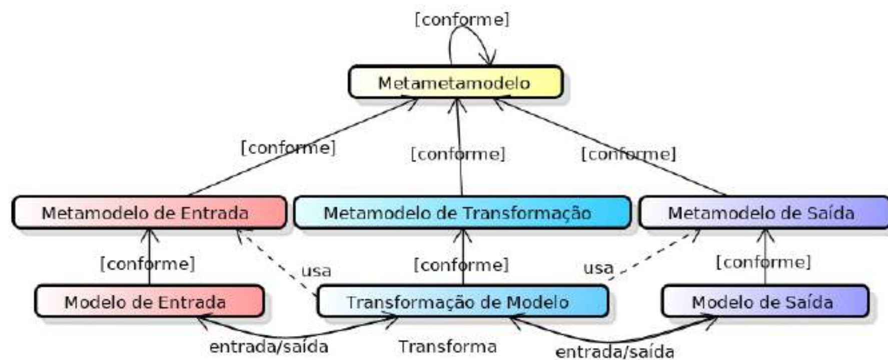


Figura 2.2: Um Esquema Representativo de Transformação de Modelos

Além dos conceitos apresentados e o esquema da Figura 2.2, Rutle et al. (2012); Mens e Van Gorp (2006) definem que:

- uma transformação é a aplicação de um conjunto de regras de transformação que descreve como o modelo de entrada pode ser transformado para o modelo de saída, seguindo um domínio de transformação;
- uma regra de transformação é uma descrição de como um ou mais construtores de uma linguagem de entrada, pode ser traduzida em uma linguagem de saída;
- um motor de transformação é uma ferramenta para aplicação ou execução de transformação de modelos.

Um modelo de transformação unidirecional contém um conjunto de regras que consiste em aceitar um ou mais modelos de entrada e produzir um ou mais modelos de saída. Por exemplo, a transformação (`Class2Relational`) de um Diagrama de Classes (DC) para o Banco de Dados Relacional (BDR) pode ser representada $DC \rightarrow BDR$. Para o mesmo caso, a transformação bidirecional pode ser definida para ambas as direções, da

esquerda para direita e vice-versa $DC \leftrightarrow BDR$, isto significa que o conjunto de regras que processa a transformação em DC, também é capaz de processar a transformação em BDR (Macedo, 2014; Mens e Van Gorp, 2006).

Diferentes tipos de TM são classificados, considerando o domínio de transformação. Em cenários em que um modelo de entrada e um de saída são suficientes, as transformações com tais características são classificadas em *um-para-um*. Porém, há domínios que requerem múltiplos modelos de entrada e de saída, nestes casos as classificações para as TMs incluem *muitos-para-muitos*, *uma-para-muitos*, ou ainda *muitos-para-um* (Brambilla et al., 2017; Rutle et al., 2012; Mens e Van Gorp, 2006). Na Tabela 2.1, são apresentados outras classificações relativas à TM, as quais estão organizadas nas colunas Classificação (nome da classificação) e a Característica (resumo das características da classificação).

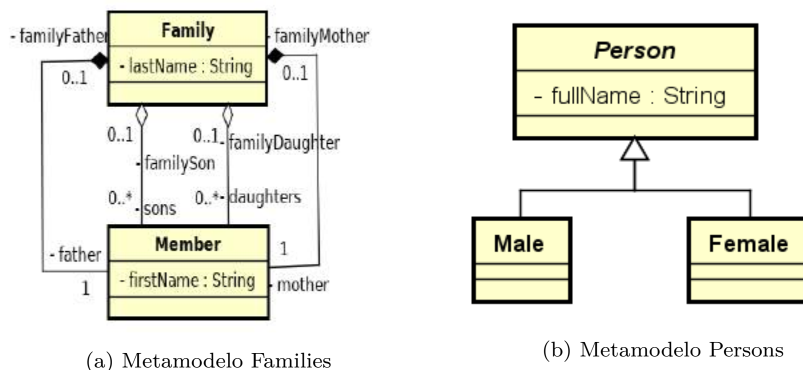
Tabela 2.1: Classificação de Transformação de Modelos

Classificação	**Característica**
Endógeno/Homogêneo	Transformações usando modelos expressados em uma mesma linguagem.
Exógeno/Heterogêneo	Transformações entre modelos expressados em diferentes linguagens.
Horizontal	Neste tipo de transformação, os modelos de entrada e de saída estão localizados no mesmo nível de abstração.
Vertical	Ao contrário da classificação anterior, os modelos de entrada e saída estão posicionados em diferentes níveis de abstração.
In-place	Em transformações *In-place*, os metamodelos de entrada e saída são coincidentes e envolvem a manipulação de um modelo, ao invés de criar um novo modelo.
Out-place	Este tipo de transformação gera um modelo de saída a partir do zero. Ela carrega o modelo de entrada e produz o modelo de saída, aplicando as regras de transformação.

As classificações de transformações apresentadas na Tabela 2.1 são relacionadas à abordagem Modelo-para-Modelo (*M2M*). Há outras abordagens, como Modelo-para-Texto (*M2T*) e Texto-para-Modelo (*T2M)*. Uma instância de TM é especificada por uma linguagem de transformação, cujas regras de transformação são criadas para determinar o domínio da transformação. Há uma grande variedade de linguagens e ferramentas para transformação de modelos. Por exemplo, Kahani et al. (2019) classificaram 60 ferramentas para este propósito, entre elas: a GrGen para transformação de grafos (Jakumeit et al., 2010); UML-RSDS, uma ferramenta MDE para propósito geral (Lano e Kolahdouz-Rahimi, 2017); Kermeta, uma linguagem imperativa (Drey et al., 2017); QVT-R, uma linguagem declarativa (OMG, 2016a); ETL (Kolovos et al., 2008), uma linguagem híbrida baseada em OCL (OMG, 2014) e implementada sob a plataforma Epsilon (Eclipse, 2014) de gerenciamento de modelos; Linguagem de Transformação Janus (JTL) (Cicchetti et al., 2011), uma linguagem para transformação bidirecional de modelos; entre outras DSLs. Além da transformação, essas ferramentas podem ser utilizadas para desenvolver, juntar, comutar, comparar e verificar modelos e metamodelos (Kahani et al., 2019). Nesta tese, a ATL é escolhida para demonstrar uma instancia de TM por meio de um exemplo de transformação de famílias em pessoas (`Families2Persons`). Os modelos `Families` e `Persons` são utilizados ao longo desta tese em exemplos de manipulação e transformação de modelos. O modelo `Families` é tido como modelo de entrada enquanto que o modelo `Pearsons` é o modelo de saída em transformações M2M.

2.1.2 Linguagem de Transformação ATL

A ATL é uma das principais linguagens usada para especificar transformação de modelos. Essa linguagem é baseada na especificação QVT (OMG, 2016a) e construída sob o formalismo da OCL (OMG, 2014) como um subconjunto da plataforma AMMA (*ATLAS Model Management Architecture*) (Bézivin et al., 2005). Kahani et al. (2019) afirmam que a familiaridade da ATL com o formalismo da OCL faz dela a linguagem mais acessível para a MDE. A ATL é uma linguagem híbrida, permitindo o uso de construtores declarativos e imperativos nas especificações de transformação de modelos (Jouault e Kurtev, 2005; Jouault et al., 2008; Eclipse, 2019a). As principais características da ATL são apresentadas com um exemplo de transformação `Families2Persons` (Eclipse, 2019b). Os Metamodelos `Families` e `Persons` estão representados respectivamente nas Sub figuras 2.3(a) e 2.3(b). A classe `Family` tem o atributo `lastName` e as agregações `familyFather`, `familyMother`, `familySon`, `familyDaughter` para a Classe `Member`, que possui o atributo `firstName`. Conforme o Metamodelo Families, é possível instanciar um modelo que contenha uma ou mais famílias em que os membros são pais e mães e pode conter filho(s) e/ou filha(s). Cada membro de uma família terá um nome e o mesmo sobrenome. O metamodelo Persons tem a classe `Person` com o atributo `fullName` e as subclasses `Male` e `Female`. Uma instancia desse Metamodelo requer que cada pessoa tenha um nome completo e seja do gênero Masculino ou Feminino.



(a) Metamodelo Families

(b) Metamodelo Persons

Figura 2.3: Metamodelos Families e Persons

Nas Listagens 2.1 e 2.2, são apresentados fragmentos do Metamodelo Families e de uma instancia desse Metamodelo, ambos expressados no formato XMI. Este domínio de transformação e os respectivos artefatos (Metamodelos e Modelo) são usados em um exemplo representativo de transformação de modelo em ATL.

Listagem 2.1: Metamodelo `Families`

```
<xmi:XMI xmi:version="2.0"
 xmlns:ecore="http://www.eclipse.org/emf/">
 <ecore:EPackage name="Families">
  <eClassifiers
      xsi:type="ecore:EClass"name="Family">
  <eStructuralFeatures
      xsi:type="ecore:EAttribute"
      name="lastName" ordered="false"
    ...
   </ecore:EPackage>
</xmi:XMI>
```

Listagem 2.2: Modelo `Families`

```
<?xml version="2.0" encoding="ISO-8859-1"?>
<xmi:XMI xmlns="Families">
<Family lastName="March">
  <father firstName="Jim"/>
  <mother firstName="Cindy"/>
  <sons firstName="Brandon"/>
 </Family>
  <Family lastName="Sailor">
    <father firstName="Peter"/>
        ...
 <\xmi:XMI>
```

As transformações em ATL são unidirecionais e aplicadas aos modelos de entrada em modo *read-only* e produzem modelos de saída em modo *write-only*. A ATL permite

regras no estilo declarativo. Regras declarativas abstraem o relacionamento entre os elementos de entrada e de saída, ocultando a semântica relacionada ao disparo de regras, ordenação e rastreabilidade. No entanto, as regras podem ser incrementadas com seções imperativas para simplificar expressões complexas (Benelallam, 2016; Eclipse, 2019a). A especificação de uma transformação de modelos em ATL é estruturada em módulos. Um módulo contêm uma seção *header* mandatória, seção `import`, `helpers` e as regras de transformações (Jouault e Kurtev, 2005; Jouault et al., 2008). A seção `header` fornece o nome do módulo de transformação (e.g., `Families2Persons`) e a declaração dos modelos de entrada e saída. A seguir, um exemplo de seção `header` é fornecido.

```
module Families2Persons;
create OUT : Persons from IN : Families;
```

Helpers permitem a definição de expressões reusáveis em OCL, os quais devem ser anexados a um contexto, que pode ser um tipo ou de um contexto global. Na Listagem 2.3, é ilustrada a implementação do `helper isFemale`. Esse `helper` fornece um contexto (`Member`) e define um tipo (`Boolean`) para tratar as referências `familyDaughter` (line 2) ou `familyMother` (line 4) como retorno `true` ou `false`. Há dois tipos de *helpers* (Jouault e Kurtev, 2005):

- operação: *helpers* de operação (Listagem 2.3) definem operações no contexto de um elemento de modelo ou no contexto de um módulo. Tais *helpers*, podem ter parâmetros de entrada e usar recursividade;
- atributo: *helpers* de atributo são usados para associar valores de atributos aos elementos dos modelos de entrada em modo *read-only*. Similar aos *helpers* de operação, esses *helpers* têm um nome, um contexto, um tipo e não aceitam parâmetros de entrada.

Os *helpers* definem restrições, padrões ou filtros para as regras. Por exemplo, na regra `Member2Male` (Listagem 2.4) a cláusula `from` aciona o `helper isFemale` para filtrar os membros de cada família do gênero masculino. Neste caso, somente elementos de entrada que satisfaçam a condição do filtro são aceitos.

Listagem 2.3: `Helper isFemale`

```
1 helper context Families!Member def: isFemale(): Boolean =
2  if not self.familyMother.oclIsUndefined() then
3   true
4   else if not self.familyDaughter.oclIsUndefined() then
5    true
6   else
7    false
8   endif
9  endif;
```

A ATL admite duas categorias de regras, *matched* e *inheritance*. Na Listagem 2.4, há um exemplo de regra *matched* (`Member2Male`). Regras *matched* em ATL são compostas por um padrão de entrada e outro de saída. Ambos podem conter um ou mais elementos correspondentes ao padrão estabelecido. Os padrões de entrada são acionados automaticamente quando uma instancia do padrão de entrada (um *match*) é identificada, que em seguida produz uma instancia de saída do padrão correspondente. A partir do *match*, implicitamente são formadas rastreabilidades transientes para associar os elementos do modelo de entrada com os elementos correspondentes do modelo de saída (Benelallam, 2016; Eclipse, 2019a). Por exemplo, a regra `Member2Male` recebe os membros de

cada família (`from s: Families!Member`, linha 3) que são filtrados pelo `helper isFemale` (`not s.isFemale()`), separando os membros de gênero masculino dos de gênero feminino. Assim, os elementos de entrada que satisfazem esse padrão (*match*) são associados ao modelo de saída no que diz respeito às pessoas do gênero masculino (`to t: Persons!Male`). O nome completo para essas pessoas é definido pela concatenação do atributo `firstName` com o resultado da execução do `helper familyName` assinalados ao atributo `fullName` (linha 6). O `helper familyName` não foi apresentado, mas ele é responsável por retornar o sobre nome (`lastName`) de cada membro da família a partir do modelo de entrada.

A conclusão da transformação de famílias para pessoas (`Families2Persons`) requer uma outra regra com os mesmos princípios da regra `Member2Male`. Neste caso, `Member2Female` para obter somente os membros do gênero feminino, transformando esses elementos em pessoas desse mesmo gênero.

Listagem 2.4: Regra Member2Male

```
1 rule Member2Male {
2   from
3     s: Families!Member (not s.isFemale())
4   to
5     t: Persons!Male (
6       fullName <- s.firstName + ' ' + s.familyName )
7 }
```

Regras *matched* em ATL são executadas em duas fases, *match* e *apply*. Na primeira fase, as regras são aplicadas sobre os elementos do modelo de entrada para satisfazer as condições de guarda. Cada *match* corresponde a criação de um link de rastreabilidade. Esse link conecta três itens: a regra que disparou a transformação, o *match* e os elementos de saída recém-criados (de acordo com o padrão de destino). Nessa fase, somente o tipo de elemento do padrão de saída é considerado, enquanto que a avaliação dos links é executada na fase seguinte. A segunda fase trata da inicialização dos recursos dos elementos de saída que estão associados a um link do elemento relacionado ao padrão de saída estabelecido. A inicialização dos recursos é realizada em duas etapas, primeiro a expressão de ligação correspondente é calculada resultando em uma coleção de elementos, e em seguida, essa coleção é passada para um algoritmo de resolução (`resolve`) para atualização final do modelo de saída (Benelallam, 2016; Eclipse, 2019a).

Na listagem 2.5, é mostrado um exemplo do resultado da transformação do modelo de entrada `Families` (Listagem 2.2) para o modelo de saída `Pearsons` (`Families2Persons`). Nessa listagem, os elementos do modelo de saída `Persons` são mostrados em formato XMI, como um desfecho do exemplo de transformação em ATL abordado nesta seção.

Listagem 2.5: Modelo de Saída `Persons`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="Persons">
      <Male fullName="Jim March"/>
      <Male fullName="Brandon March"/>
      <Male fullName="Peter Sailor"/>
      <Male fullName="David Sailor"/>
      <Male fullName="Dylan Sailor"/>
      <Female fullName="Cindy March"/>
          ...
<\xmi:XMI>
```

Nesta seção foram apresentados os principais conceitos sobre transformação de modelos, como também uma visão geral sobre a abordagem MDE. Um exemplo de transformação de modelos foi apresentado para ilustrar as principais características da ATL. Na próxima seção, a abordagem orientada a dados (Dc) é apresentada.

2.2 ABORDAGEM ORIENTADA A DADOS

O termo *Data centric* foi usado por Breitbart et al. (1993) para separar as abordagens transacionais envolvendo vários bancos de dados, como também nas operações de dados que se intercalam para produzir uma execução correta. Para Zu et al. (2016), Dc é uma abordagem que provê uma maneira natural e simplificada de visualizar e modelar aplicações distribuídas. Uma abordagem Dc se concentra nos dados que se movem e se transformam no sistema, e não nos processos que executam essas operações. Qin et al. (2016) usaram os aspectos da Dc para revisar as principais técnicas e os esforços de pesquisa em IoT, incluindo processamento de fluxo de dados, modelos de armazenamento de dados e o processamento de eventos. No trabalho de (Kambatla et al., 2014), a Dc foi interpretada como uma abordagem que: permite o envio de tarefas a serem processadas em um ambiente computacional, possui modelos de programação, absorve carga de dados e processamento paralelo/distribuído de dados.

Aplicações orientada a dados são consideradas aptas ao gerenciamento de informações de diferentes fontes de dados e formatos, incluindo dados armazenados em bancos de dados. Aplicações Dc deixam de ser úteis para um conjunto restrito de domínios e se tornam importantes para domínios como sistemas de rede e sistemas distribuídos, processamento de linguagem natural, análise de compiladores, robótica modular, segurança, entre outros. Além disso, a abordagem Dc ganha espaço além do paralelismo de dados orientado a lote (*batch*), incluindo as linguagens declarativas orientada a dados (Alvaro et al., 2010).

Associada a um modelo de dados e a um SGBD, a SQL é uma linguagem declarativa comumente utilizada para a manipulação de dados. Essa linguagem é considerada a principal linguagem Dc. Além da SQL, há linguagens como a Bloom (Alvaro et al., 2011; BOOM, 2013) projetada para o desenvolvimento de aplicações Dc. Tal linguagem permite a manipulação de dados com expressões declarativas sobre coleções de dados de maneira paralela/distribuída. Embora o estilo declarativo seja predominante em aplicações Dc, na linguagem Bamboo (Zhou e Demsky, 2010) as tarefas de manipulação de dados são escritas em orientação a objetos. Na maior parte dos casos, as linguagens Dc são aplicadas para propósito geral, permitindo operações de dados em alto nível. Essas linguagens permitem que os aspectos associados a distribuição de dados, escalonamento, balanceamento de carga e comunicação entre os processos sejam tratados pelo sistema em tempo de execução. Assim, linguagens Dc podem ser adaptadas para várias formas de computação paralela/distribuída incluindo *clusters*, computação em grade e computação em nuvem (Davidson et al., 2006; Zhou e Demsky, 2010; Alvaro et al., 2010). Em razão disto, as aplicações em Dc favorecem o uso de programação paralela implícita, em que o(a) desenvolvedor(a) não precisa especificar primitivas relacionadas ao paralelismo, podendo ampliar o foco no domínio da aplicação.

Implementações de aplicações paralelas impõem desafios, sejam implementações por meio das abordagens de programação paralela implícita ou explicita. Ambas diferem no nível de automação do paralelismo e na garantia da eficiência. As abordagens implícitas dependem das linguagens paralelas e estruturas para facilitar a implementação, ocultando

complexidades internas relacionadas ao paralelismo. Por outro lado, as abordagens explícitas assumem que o(a) desenvolvedor(a) é capaz de decompor tarefas e coordená-las em nível de programação. Embora o nível de abstração do paralelismo seja menor em relação às abordagens implícitas, as abordagens explícitas tendem a oferecer mais flexibilidade ao desenvolvedor(a), o que às vezes pode proporcionar um melhor desempenho às aplicações oriundas dessas abordagens (Rauber e Rnger, 2013).

As linguagens funcionais também são atrativas para implementação de aplicações paralelas. Essas linguagens oferecem técnicas de abstração que podem ser usadas na computação e coordenação de tarefas concorrentes. Além disso, a imutabilidade aplicada a estrutura de dados elimina dependências desnecessárias, comumente presentes em aplicações paralelas. Linguagens como Scala[3], Clojure[4], Haskell[5] e Elixir[6] são exemplos de linguagens funcionais. Tais linguagens, em conjunto com frameworks de processamento distribuído (e.g., como o Hadoop[7]) propiciam o desenvolvimento de aplicações implicitamente paralela/distribuída. Por exemplo, a Clojure implementada sob o framework Hadoop, Elixir compilada para a Máquina Virtual Erlang[8], utilização da biblioteca de concorrência da linguagem Haskell, entre outras implementações.

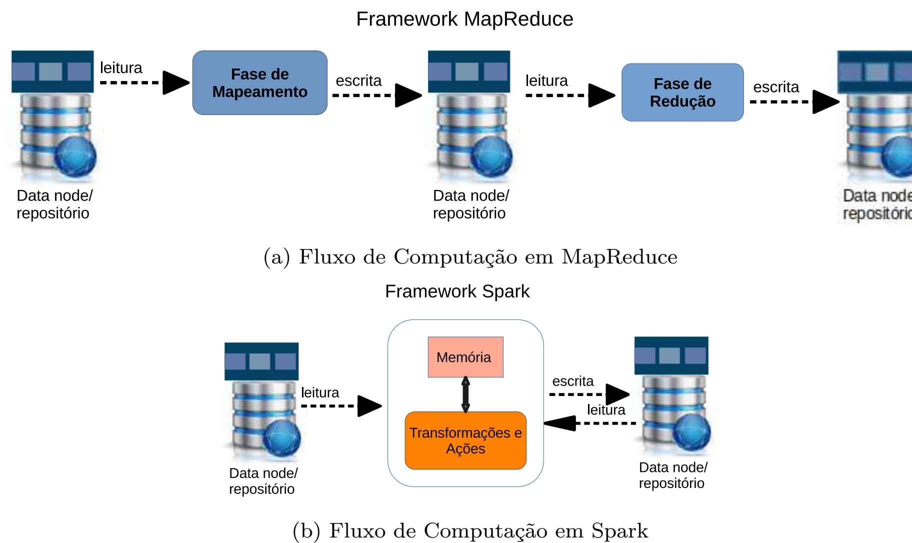
Modelos de programação paralela/distribuída também têm sido utilizados para o desenvolvimento de aplicações dessa natureza. O modelo MapReduce é um dos exemplos bem sucedidos de paralelismo implícito. Esse modelo foi desenvolvido para aplicações paralela/distribuída de propósito geral pautado nas funções `Map` e `Reduce`, as quais são definidas pelo(a) desenvolvedor(a). A semântica dessas funções possibilita o paralelismo de dados, facilitando o desenvolvimento de aplicações direcionadas ao processamento intensivo de dados (Dean e Ghemawat, 2008; Benelallam, 2016). Além do modelo MapReduce, há frameworks que oferecem APIs para o desenvolvimento de aplicações distribuídas. O Spark (Apache, 2019d) é um framework de uso geral para o desenvolvimento de aplicações paralelas/distribuídas, as quais podem ser executadas em um *cluster* de nós (máquinas) ou utilizando núcleos de processador(es) de uma única máquina.

A abordagem orientada a dados foi escolhida para compor a Dc4MT com o propósito de suprir os aspectos como, paralelismo implícito, programação declarativa em alto nível e escalabilidade, que comumente não estão presentes nas abordagens tradicionais de transformação de modelos, como a ATL, Kermeta, Viatra, entre outras (Tisi et al., 2013; Burgueño et al., 2016; Benelallam et al., 2016). Além disso, aplicações orientada a dados têm como foco principal o processamento de dados. Os esforços de tais aplicações concentram-se em operações de movimentação e transformação dos dados no sistema, e não no modo como essas operações são executadas Zu et al. (2016). O framework Spark e as suas APIs RDD, DataFrame e GraphFrames são adotados para uma implementação da abordagem Dc4MT. Essas APIs e o framework Spark são apresentadas na próxima seção.

2.2.1 Framework Spark

O Apache Spark (Apache, 2019d) é um framework de uso geral para o processamento de dados em modo paralelo/distribuído. Suas APIs permitem especificar diferentes tipos de computação, como consultas interativas, processamento de fluxo de dados, pro-

[3]https://www.scala-lang.org/
[4]https://clojure.org/
[5]https://www.haskell.org/
[6]https://elixir-lang.org/
[7]http://hadoop.apache.org/
[8]https://www.erlang.org/



(a) Fluxo de Computação em MapReduce

(b) Fluxo de Computação em Spark

Figura 2.4: Uma Visão Geral da Computação nos Frameworks MapReduce e Spark

cessamento de grafos, entre outras. Na Figura 2.4, é ilustrado uma visão simplificada do fluxo de computação dos frameworks MapReduce e Spark. Na Sub figura 2.4(a), é mostrado um exemplo da computação no modelo MapReduce, a qual é realizada pelas fases de Mapeamento e Redução alimentadas pelas operações de Leitura e Escrita. Na Sub figura 2.4(b), é ilustrado um exemplo de como a computação no Spark é realizada. As operações de Transformação e Ação são realizadas em Memória inicializadas pela operação de Leitura. Durante a execução dessas operações (Leitura ou Escrita) a memória secundária pode ser acionada, quado a memória principal não for suficiente para concluir tal execução. No Spark, uma operação de Ação pode incluir a Escrita como saída final da computação. Observa-se que na Figura 2.4, os frameworks utilizam Data node (HDFS - *Hadoop Distributed File System*) ou repositório como representação de instância de dados para as operações de Leitura ou Escrita. Porém, outras opções de modelos de instâncias de dados, como os modelos NoSQL ou relacional podem ser conectados a esses frameworks.

Os frameworks ilustrados na Figura 2.4 fornecem paralelismo implícito, mas a computação de operações recorrem à abordagens diferentes. No MapReduce/Hadoop a computação é baseada nas funções de Mapeamento e Redução com leituras e escritas entre elas. No Spark, a computação é direcionada para a memória principal e dividida em dois tipos: Transformação, cujo resultado é armazenado em memória principal e Ação, que pode conter diretivas de escrita para memória secundária. Um exemplo de transformação é a carga de dados contidas em diferentes fontes (repositórios, HDFS, NoSQL, entre outras) para processamento em memória principal, evitando que esses dados sejam gravados em disco ao final de cada iteração do processamento, reduzindo o tempo gasto com escrita e leitura na memória secundária (Zomaya e Sakr, 2017).

O framework Spark oferece um arcabouço com um conjunto de bibliotecas e APIs. Incluem nesse conjunto, bibliotecas para a SQL, dados estruturados, dados semi-estruturados, aprendizagem de máquina, processamento de *stream* e grafos. As APIs DataFrame e GraphFrames permitem a manipulação de dados em alto nível, enquanto que a RDD (*Resilient Distributed Dataset*) é considerada de baixo nível. Além de diferentes níveis de abstração, providos por tais APIs, o(a) desenvolvedor(a) pode recorrer a manipulação de RDDs ou ao uso de variáveis compartilhadas como a `broadcast` e `accumulators` no desenvolvimento de aplicações distribuídas (Apache, 2019d; Chambers e Zaharia,

2018; Zomaya e Sakr, 2017). A API RDD fornece mecanismos de distribuição de dados (particionamento), garantindo o processamento resiliente de dados. As APIs DataFrame e Graphframes abstraem da API RDD tais mecanismos (particionamento e resiliência).

Na Figura 2.5, a arquitetura do framework Spark é ilustrada. O Nó Master é o responsável por controlar a execução de uma aplicação Spark, mantendo de forma independente um conjunto de processos em nós (*cores*) de uma máquina ou em um cluster de máquinas (Nó Worker). A execução é coordenada pelo objeto Contexto Spark contido no Programa Driver. Tal objeto, troca mensagens com o Gerenciador de Clusters, que pode ser um gerenciador independente do Spark, como o YARN ou Mesos para obter recursos e inicializar os Executores (nós Workers). Um Executor tem o papel de receber as Tarefas (pequenas unidades de execução) atribuídas pelo Nó Master, executá-las e reportá-las de volta com os respectivos estados e resultados. A interação entre os nós Workers e o Contexto Spark é suportada pelo Gerenciador de Clusters responsável por manter um Cluster de máquinas ou de nós em operação durante a execução de uma ou mais aplicações Spark. Um nó Worker é qualquer nó com condições de executar um código de aplicação Spark. Além disso, ele pode armazenar dados em cache, ler e escrever dados de e para fontes externas (Chambers e Zaharia, 2018; Apache, 2019d).

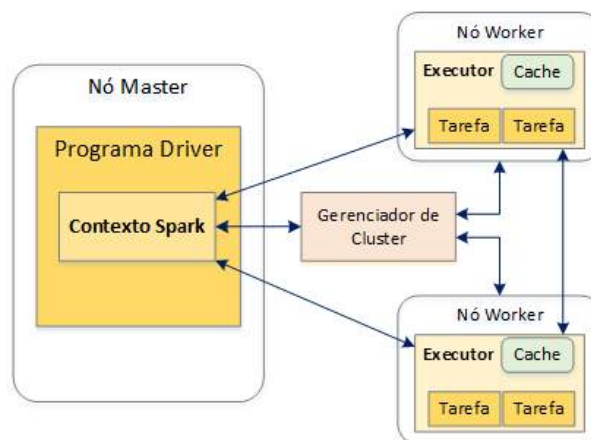


Figura 2.5: Arquitetura do Spark em Modo Cluster com Três Nós

A tolerância a falhas e o paralelismo em operações sobre dados, utilizando *clusters* são apoiadas pelos RDDs. Um RDD representa uma coleção imutável de elementos que pode ser manipulada em paralelo. Um dado em RDD é uma instância de um objeto, diferente de uma estrutura formada por linhas e colunas como em um DataFrame. Sobre os RDDs são aplicadas operações de Ações ou Transformações, resultando em novos RDDs. Em situação de falha, a cadeia de RDDs pode ser reconstruída a partir do último RDD disponível, aplicando as mesmas operações. Isto é possível porque cada RDD tem a capacidade de prover a linhagem de dados, isto significa que o conceito de *Data lineage* (Tang et al., 2019) está presente na API RDD. Operações de transformação podem possuir diversas RDDs como entrada ou saída, além de realizar a comunicação de dados entre vários nós, visto que as RDDs podem possuir instâncias em diferentes nós de um ambiente distribuído. Uma operação de Transformação é aplicada para criar uma nova RDD no Nó Worker (e.g., `spark.read.format("json")`), já a Ação é uma operação que instrui o framework Spark a realizar a computação no Nó Worker e passar o resultado de volta para o Nó Master (e.g., `write.json("file.json")`) (Vasata, 2018; Apache, 2019d; Chambers e Zaharia, 2018). De modo geral, considera-se as atribuições ao acrônimo RDD como:

- *Resilient*, tolerante a falhas e é capaz de recuperar dados decorrentes de falhas;
- *Distributed*, distribui dados entre múltiplos nós em um *cluster* ou de uma máquina;
- *Dataset*, coleções de dados particionados com valores.

Os RDDs são usados como base para as execuções paralelas de tarefas pelas as APIs Dataframe e GraphFrames. Por exemplo, em execuções paralelas a API DataFrame recorre a API Dataset, uma vez que essa API fornece estruturas de dados tipados e organizados em coleções. Além disso, a API Dataset permite a definição de coleções estruturadas para receber dados de objetos da JVM (*Java Virtual Machine*)[9], cuja manipulação se dá por meio de transformações funcionais como `map`, `flatMap`, `filter`, entre outras.

Um DataFrame é estruturado em um Array bi-dimensional, em que cada coluna pode conter valores e cada linha possui um conjunto de valores para cada coluna, formando um registro de dados do tipo `Row`. Um DataFrame pode receber dados agrupados em colunas, as quais estão acessíveis via o próprio esquema. Um DataFrame pode ser transformado em novos DataFrames por meio de operadores relacionais em expressões baseadas em funções SQL. DataFrames e Datasets são coleções (*table-like*) definidas por linhas e colunas. Cada coluna tem o mesmo número de linhas, e cada coluna tem um tipo de dado que deve ser consistente para cada linha da coleção (Apache, 2019d; Chambers e Zaharia, 2018). Na Figura 2.6, é ilustrado um exemplo de DataFrame estruturado em três linhas e cinco colunas. Os dados contidos nesse DataFrame são extraídos do modelo `Families` (linhas com `March, Sailor, and Camargo families`). Uma coluna é definida com um tipo de dado, que pode ser `String, Integer, Date, Boolean` ou `Array`.

Colunas

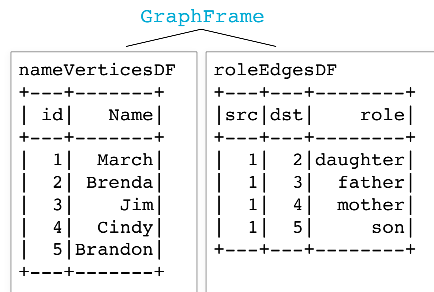
Linhas

lastName	daughters	father	mother	sons
March	[[, Brenda]]	[, Jim]	[, Cindy]	[[, Brandon]]
Sailor	[[, Kelly]]	[, Peter]	[, Jackie]	[[,David],[,Dy...
Camargo	[[, Jor], [, Teste]]	[, Luiz]	[, Sid]	[[, Lucas]]

Figura 2.6: Um Exemplo de DataFrame

A API GraphFrames é usada para processar dados em formato de grafo, o qual é formado pela combinação de dois DataFrames. Na Figura 2.7, há um exemplo ilustrativo de uma instância da família March em um GraphFrames. Essa instância é criada a partir dos Dataframes `nameVerticesDF` e `roleEdgesDF`, os quais representam respectivamente os vértices e arestas de um grafo. Para a formação de um GraphFrames é indispensável que o DataFrame que representa os vértices contenha uma coluna com o nome `"id"` para especificar um único ID para cada vértice do grafo. Já o DataFrame que representa as arestas do grafo precisa ter a especificação de duas colunas: a `"src"`, como o vértice de origem e a `“dst”`, como o vértice de destino, formando assim uma aresta. Dependendo do domínio de aplicação, outras colunas podem compor os Dataframes que darão origem aos vértices e arestas do grafo (Chambers e Zaharia, 2018; Dave et al., 2016). Por exemplo, no DataFrame `nameVerticesDF` a coluna `name` é usada para receber o nome de cada membro da Família, incluindo o sobrenome March (vértice `1`). Da mesma forma, o DataFrame `roleEdgesDF` tem a coluna `role` para receber o papel de cada membro da família March.

[9]O framework Spark é implementado na linguagem Scala e utiliza a JVM



id	Name
1	March
2	Brenda
3	Jim
4	Cindy
5	Brandon

src	dst	role
1	2	daughter
1	3	father
1	4	mother
1	5	son

Figura 2.7: Família March em um GraphFrames

A API GraphFrames provê as mesmas operações que a API DataFrame, tais como `map, select, filter, join`, entre outras. Há um conjunto de algoritmos embutido na GraphFrames especificamente para manipulação de grafos, como o BFS (*Breadth-First Search*), *Label Propagation*, *PageRank*, e outros. Os conceitos de imutabilidade e tolerância a falhas presentes na API RDD são abstraídos pelas APIs GraphFrames and DataFrame, assim como a avaliação tardia (*lazy-evaluation*). Isso significa que o Spark executa instruções das operações o mais tarde possível, utilizando um Grafo Direcionado Acíclico (DGA *Directed Acyclic Graph*) em vez de modificar os dados imediatamente quando uma operação é executada (Dave et al., 2016; Chambers e Zaharia, 2018).

O paralelismo implícito e os recursos disponíveis no Spark, podem simplificar a distribuição de dados e tarefas. No entanto, o processamento de dados semi-estruturados (modelos) pode apresentar desafios diferentes em relação ao processamento de dados simples, como um arquivo estruturado, por exemplo. Como nesta tese, busca-se o processamento paralelo/distribuído de modelos, que geralmente são densamente interconectados, na próxima seção técnicas de particionamento de modelos são apresentadas.

2.3 PARTICIONAMENTO DE MODELOS

Particionamento de modelos tem sido usado para vários propósitos, principalmente quando o modelo a ser processado é constituído por milhares de elementos. O particionamento de modelos é uma operação que consiste em extrair de um modelo um conjunto de submodelos, dividindo esse modelo em fragmentos, sem perder a consistência (Benelallam et al., 2016; Wei et al., 2016; Hartmann et al., 2015; Blouin et al., 2015; Garmendia et al., 2014; Ujhelyi et al., 2012; Lano e Kolahdouz-Rahimi, 2010). Por exemplo, dado um modelo M, esse modelo pode ser fragmentado em um conjunto de fragmentos $\{F_1, F_2..., F_n\}$ ($M \subset \{F_1, F_2..., F_n\}$), sendo que cada fragmento F contem um ou mais elementos E_M do modelo, mas um elemento não pode estar em mais de um fragmento ($E_M \notin \{F_1, F_2..., F_n\}$). Além disso, os elementos podem conter links entre si. Links intra-fragmentos referenciam elementos que estão em um único fragmento e links inter-fragmentos, são aqueles que referenciam elementos que estão em diferentes fragmentos (Scheidgen et al., 2012). Por ser uma operação complexa, há várias estratégias que podem ser adotadas para o particionamento de modelos, tais como o uso de anotações, particionamento de grafos e o fatiamento.

2.3.1 Anotações

Uma anotação é uma informação extra que pode ser adicionada em artefatos de software, documentos, entre outros. Por exemplo, anotações podem ser adicionadas a me-

tamodelos para acrescentar uma forma especial de sintaxe; podem ser inseridas em códigos fontes para prover metadados (Rocha e Valente, 2011). Anotações em modelos são usadas em operações de transformações de modelos, mas elas podem ajudar no particionamento de modelos, identificando e ordenando partes do modelo sob particionamento (Macedo, 2014). Anotações podem ser úteis para mapear determinados pontos do modelo, e esses pontos podem receber diferentes tratamentos durante o particionamento (Khalek, 2011). No entanto, somente as anotações não são suficientes ao particionamento de modelos, é necessário adotar alguma técnica de particionamento.

2.3.2 Particionamento de Grafos

Um grafo pode ser representado como $G(V, E)$, em que o V corresponde a um conjunto de vértices e E a um conjunto de arestas do grafo G. Um subgrafo S_G de um grafo G é um grafo, cujos vértices $V(S_G)$ são um subconjunto do conjunto de vértices $V(G)$, em que $V(S_G) \subseteq V(G)$ e o conjunto de arestas $E(S_G)$ é um subconjunto das arestas em $E(G)$.

Frequentemente usados na computação para representar problemas complexos, grafos são úteis para modelar problemas computacionais. O particionamento de grafo serve para diminuir a complexidade de um problema, considerando que esse grafo é particionado em subgrafos (o problema é decomposto em partes menores) e pode ser processado em paralelo. O particionamento de grafos tratado nesta seção é adotado no âmbito de modelagem e transformação de modelos, uma vez que há a utilização de grafos em vários contextos. Grafos podem ser usados em TMs para traduzir as instâncias de uma linguagem de modelagem para outra em um processo de TM, uma vez que as estruturas dessa linguagem podem ser representada por um tipo de grafo. A Gramática de Triplo Grafos (TGGs, *Triple Graph Grammars*) (Schürr, 1995) é considerada como uma linguagem para TM baseadas em grafos (Tomaszek et al., 2018). A expressividade, estilo declarativo e o formalismo presentes nessa linguagem são aspectos relevantes para a transformação de grafos, dado que um conjunto de regras em triplas é suficiente para gerar regras de transformação bidirecional de modelos (Hermann et al., 2014). Além disso, a gramática TGGs permite o mapeamento de relacionamentos entre as instâncias dos modelos em pares, o que pode favorecer o particionamento desses modelos.

Para o particionamento de grafos, há um conjunto de heurísticas e métodos que buscam dividir um grafo em subgrafos. Por exemplo, Deák et al. (2013) propuseram um método para o particionamento de grafos em três fases e para cada uma delas uma heurística é adotada. Na primeira fase, os vértices do grafo (nós) são separados aleatoriamente em partições. Em seguida, as partições são subdividas em subpartições com a aplicação de algoritmos de expansão. Na terceira e última fase, as partições são submetidas a algoritmos de refinamento. Benelallam et al. (2016) consideram que modelos podem ser representados em formato de grafos, dessa perspectiva eles propuseram uma estratégia para o particionamento de modelos. A partir das regras de transformação, a dependência entre os elementos do modelo é mapeada e em seguida um algoritmo do tipo *greedy*[10] é aplicado visando a distribuição balanceada de dados (partições) durante o processo de transformação de modelos.

O modelo de particionamento proposto por Benelallam et al. (2016) pode servir de base para abordagens que buscam o particionamento de dados semi-estruturados com o balanceamento entre as partições. Partições de dados com tamanhos semelhantes podem

[10]ttps://xlinux.nist.gov/dads//HTML/greedyalgo.html

favorecer o desempenho do processamento dessas partições em ambientes distribuídos. Além do particionamento de dados usando heurísticas, há outras técnicas que podem ser úteis ao particionamento de modelos, como a técnica de fatiamento (*slice*).

2.3.3 Técnica *Slice*

A técnica Slice foi criada para a otimização de programas imperativos, reduzindo o tamanho do programa sem alterar o seu comportamento (Weiser, 1981). Essa técnica é usada como base para o particionamento de programas e modelos. Uzuncaova e Khurshid (2007) propuseram a técnica Slice Declarativa para a otimização de modelos declarativos especificados em Alloy (Jackson e Torlak, 2018). Essa técnica apresenta ganhos de desempenho em relação a forma tradicional de analisar modelos, mas o modelo é dividido em apenas dois submodelos. Ganov et al. (2011) estenderam a técnica Slice Declarativa, permitindo o particionamento de modelos declarativos em múltiplas partes. Nessa extensão, é usado um procedimento recursivo para dividir o modelo, baseado em critérios de corte, em que o menor critério é aquele que possui o mínimo de variáveis livres. Inclui nesse procedimento um grafo de dependência entre os critérios de corte. Uma vez que o grafo é criado, uma ordenação topológica é realizada resultando em um ordenado conjunto de critérios de cortes, indicando quais critérios podem ser executados em paralelo.

Um exemplo da técnica Slice no contexto da MDE: dado um diagrama de classes contendo as classes $\mathtt{Cl}_1$ e $\mathtt{Cl}_2$, ambas dependentes do pacote $\mathtt{Pc}_1$. A classe $\mathtt{Cl}_1$ possui o atributo $\mathtt{At}_1$ e a classe $\mathtt{Cl}_2$ possui os atributos $\mathtt{At}_2$ e $\mathtt{At}_3$. Os atributos ($\mathtt{At}_1$, $\mathtt{At}_2$, $\mathtt{At}_3$) são dependentes do *DataType* $\mathtt{Dt}_1$. Neste cenário, o critério de particionamento `C` é assinalado para os elementos do tipo classe, isto significa que fragmentos *slices* do diagrama de classes serão gerados com base nas classes e suas dependências. Neste caso, pode-se obter os Slices $\mathtt{Sl}_1$ com os seguintes elementos $\{\mathtt{Pc}_1,\mathtt{Cl}_1,\mathtt{Dt}_1,\mathtt{At}_1\}$ e o $\mathtt{Sl}_2$ com os elementos $\{\mathtt{Pc}_1,\mathtt{Cl}_2,\mathtt{Dt}_1,\mathtt{At}_2,\mathtt{At}_3\}$.

O particionamento de modelos busca diminuir a complexidade na manipulação de modelos, dividindo-os em partes menores, as quais são usadas para diferentes propósitos (Lano e Rahimi, 2011). Por exemplo: Shaikh et al. (2011) tornam a verificação da *satisfiability* de modelos mais eficientes por fragmentá-los; Blouin et al. (2014, 2015) propuseram o particionamento de modelos para a visualização de características específicas de modelos e metamodelos, em especial modelos grandes; Lallchandani e Mall (2011) buscam o particionamento de modelos para visualizar diferentes aspectos do sistema em nível de arquitetura.

Além da técnica Slice, outras abordagens são usadas para o particionamento de modelos. Garmendia et al. (2014) propuseram uma abordagem baseada em anotações e em conceitos de modularidade (projeto, pacote e unidade). Essa abordagem permite a edição de modelos de acordo com a estrutura de modularidade, dividindo e reconstruindo modelos monolíticos por meio de uma estratégia intercambiável. Amálio et al. (2015) propuseram um formalismo voltado a fragmentação de modelos, tendo em vista os aspectos de complexidade e escalabilidade presentes na MDE. O formalismo chamado de Fragmenta é baseado em modularidade, separação de interesse, clusterização, grafos e morfismo. Esse formalismo oferece uma visão geral de como particionar um modelo em fragmentos e organizá-los em *clusters*.

Além das estratégias, técnicas e abordagens voltadas ao particionamento de dados e modelos, o processamento em paralelo/distribuído desse tipo dados requer a atenção aos aspectos como balanceamento de carga, minimização de *data skew* e localização

de dados (Gong et al., 2019; Tang et al., 2018). Sob esses aspectos, o processamento paralelo/distribuído de modelos pode ser melhorado.

2.4 CONSIDERAÇÕES FINAIS

A abordagem Dc4MT é embasada nos principais conceitos das abordagens MDE e Dc. Esses conceitos foram apresentados neste capítulo com os níveis de abstração de modelos providos pela MDE. Também foram apresentados os conceitos e tipos de transformação de modelos seguido por um exemplo de especificação de uma TM em ATL. As características da ATL como *matching* e *helpers* foram apresentadas, uma vez que elas se assemelham com as construções de filtros e regras especificadas para abordagem Dc4MT.

Os aspectos da abordagem Dc como paralelismo implícito e construções declarativas são apresentados como uma estratégia para o desenvolvimento de aplicações paralela/distribuída. Um conjunto de tecnologias foi apresentado como alternativas para desenvolvimento de aplicações escaláveis, sobretudo a transformação de modelos. Entre as ferramentas para o desenvolvimento de transformações de modelos com a probabilidade de processar VLMs, foram selecionados o framework Spark e as suas APIs RDD, DataFrame e GraphFrames. As características dessas tecnologias favorecem a implementação a que se propõem para a abordagem Dc4MT. No entanto, a Dc4MT pode ser implementada com outras ferramentas de paralelismo implícito (e.g., MapReduce/Hadoop).

As estratégias de particionamento de modelos baseadas em conceitos de anotações de modelos, grafos no âmbito de modelagem e a técnica *Slicing* foram apresentadas neste capítulo como alternativas que servem de base para a fragmentação e o particionamento de modelos. No próximo capítulo, trabalhos relacionados a esta tese são apresentados.

3 TRABALHOS RELACIONADOS

Um conjunto expressivo de trabalhos relacionados à transformação de modelos (TM) tem sido publicado, exemplo disso são os *surveys*: Kahani et al. (2019); Lano et al. (2018); da Silva (2015); Mens (2013), entre outros. Esses *surveys* descrevem técnicas, abordagens, ferramentas, conceitos, linguagens, atividades e outros aspectos pertinentes à transformação de modelos. Kahani et al. (2019) classificam e organizam em diferentes perspectivas 60 ferramentas destinadas à transformação de modelos, fornecendo um panorama atual de possibilidades no que diz respeito ao ferramental e abordagens. Também há *surveys* dedicados a abordagem Data Centric (Dc), como os de Mazumdar et al. (2019); Kougka et al. (2018); Qin et al. (2016) e Caíno-Lores e Jesús (2016). Tais *surveys* abordam questões como a otimização no processamento de dados, processamento de dados em *stream*, processamento de eventos, localidade de dados em infraestrutura de larga escala, ambientes distribuídos, entre outros domínios em que a Dc é utilizada.

Os trabalhos apresentados neste capítulo estão divididos em dois grupos: o primeiro grupo contém trabalhos envolvendo a execução paralela e ou distribuída de transformação de modelos; no segundo grupo, estão os trabalhos em que a abordagem Dc é utilizada em diferentes domínios de aplicação, com foco em transformações de modelos. Esses trabalhos são extraídos de diferentes repositórios de busca e classificados de acordo com os critérios de seleção estabelecidos para o âmbito desta pesquisa. Os trabalhos selecionados e classificados são resultantes de uma revisão estruturada da bibliografia relacionada ao tema desta tese. O resultado da seleção e classificação desses dois grupos de trabalhos são apresentados nas próximas seções.

3.1 TRANSFORMAÇÃO PARALELA E/OU DISTRIBUÍDA DE MODELOS

A metodologia de seleção e classificação de trabalhos que envolvem a transformação paralela e ou distribuída de modelos está estruturada nos seguintes passos:

a) definição de um conjunto de *strings* e dos repositórios de busca;

b) definição de critérios de inclusão e exclusão de trabalhos;

c) seleção e classificação de trabalhos.

A *string* base para as buscas de trabalhos está definida na Listagem 3.1, seguida pelos endereços de quatro repositórios de busca.

Listagem 3.1: *String* de Busca - Transformação Paralela e/ou Distribuída de Modelos (TP/DM)

```
(((model-driven engineering) OR (MDE) OR (model driven engineering) OR (model transformations)
   OR (transformation rules)) AND ((distributed Computing) OR (Parallel Computing) OR
   (distributed System) OR (Parallel system) OR (scalable execution) OR (implicity parallelism)
   OR (distributed programming model) OR (distributed model transformations) OR (distributed
   environments) OR (scalable MDE) OR (scalable model transformation) OR (distributed
   transformations) OR (parallel model transformations) OR (VLM transformations) OR (LM
   transformations) OR (model partitioning) OR (model splitting) OR (model fragmentation)))
```

http://dl.acm.org
http://scholar.google.com
http://ieeexplore.ieee.org
http://www.sciencedirect.com

São considerados critérios de inclusão: trabalhos que versem à respeito da TP/DM (Transformação Paralela/Distribuída de Modelos) visando a escalabilidade; trabalhos que utilizam mecanismos para a abstração do paralelismo ou da distribuição no processamento de regras de transformação; trabalhos que incluem explicitamente alguma estratégia ou técnica de particionamento de modelos (PM). O ano de publicação dos trabalhos também compõem os critérios de inclusão e não é inferior à 2009. Trabalhos que não incluem TP/DM ou não utilizam nenhuma técnica de particionamento ou de fragmentação de modelos no processo de TM, não são selecionados (critério de exclusão).

As buscas foram executadas no período de 28/10 à 05/11/2019 e revisadas em 05/2020, utilizando os quatro repositórios escolhidos previamente e tendo como base a *string* de busca contida na Listagem 3.1. De acordo com os critérios de inclusão e exclusão, foram selecionados e classificados 17 trabalhos, os quais são estruturados a seguir:

- Abordagem: identificar no trabalho o tipo de abordagem utilizada na transformação de modelos, por exemplo a Relacional;
- Framework: indicar se o trabalho utiliza algum framework no processo de transformação de modelos;
- Linguagem: listar qual ou quais linguagens foram utilizadas para as especificações de transformação de modelos;
- Estilo: identificar qual o estilo de especificação adotado para expressar as regras de transformação de modelos;
- TP/DM: esse critério é para indicar se há execução paralela e/ou distribuída de transformação de modelos;
- PM: para apontar se há alguma estratégia de particionamento/fragmentação de modelos adotada no trabalho, não inclui o particionamento implícito do ambiente utilizado na execução da transformação.

Esses itens estão organizados em colunas nas Tabelas 3.1 e 3.2 (partes 1 e 2 da classificação), com exceção para as colunas "#" e Trabalho. A coluna "#" recebe uma numeração, relacionando a descrição de cada trabalho com a classificação contida nessas tabelas. A coluna Trabalho é destinada para exibir a referência de cada trabalho.

Tabela 3.1: Classificação de Trabalhos Relacionados à TP/DM com ou sem PM (parte 1)

#	**Trabalho**	**TP/DM**	**PM**
1	(Benelallam, 2016)	√	√
2	(Aracil e Ruiz, 2017)	√	√
3	(Benelallam et al., 2015)	√	-
4	(Burgueño, 2016)	√	-
5	(Fekete e Mezei, 2016)	√	-
6	(Benelallam et al., 2018)	√	-
7	(Tisi et al., 2013)	√	-
8	(Imre e Mezei, 2012)	√	-
9	(Daniel et al., 2017)	√	-
10	(Benelallam et al., 2016)	-	√
11	(Cuadrado e de Lara, 2013)	-	√
12	(Amálio et al., 2015)	-	√
13	(Scheidgen et al., 2012)	-	√
14	(Wei et al., 2016)	-	√
15	(Garmendia et al., 2014)	-	√
16	(Fan et al., 2020)	-	-
17	(Clasen et al., 2012)	-	-

***TP/DM** Transformação Paralela/Distribuída de Modelos. ***PM** Particionamento de Modelos

De acordo com a Tabela 3.1, os itens de classificação de trabalhos TP/DM e PM quando presentes nos trabalhos selecionados são assinalados com "√", do contrário são assinalados com "-". O trabalho Clasen et al. (2012) motivou a escolha desses itens para a classificação dos trabalhos selecionados, especificamente os itens PM/DM e PM. Os demais itens que compõem a classificação estão na Tabela 3.2. O resultado da classificação dos trabalhos do primeiro grupo são discutidos após as descrições desses trabalhos. Essas descrições estão organizadas nas subseções TP/DM com Particionamento ou Fragmentação do Modelo de Entrada, TP/DM sem Particionamento ou Fragmentação do Modelo de Entrada[1].

3.1.1 TP/DM com Particionamento ou Fragmentação do Modelo de Entrada

1. Benelallam (2016) propõem uma abordagem relacional para transformação distribuída de modelos, usando a linguagem ATL, o Framework Hadoop e o modelo MapReduce para a execução distribuída de transformações. Na junção entre a linguagem ATL e o MapReduce não há modificações na sintaxe da linguagem e nenhuma primitiva para a distribuição é adicionada, tornando implícita a execução e a distribuição das regras de transformação. O particionamento de modelos adotado nesse trabalho inclui uma análise estática das regras de transformação, seguida da aplicação de um algoritmo do tipo *greedy* como uma heurística da estratégia de particionamento do modelo de entrada. Essa estratégia busca melhorar a eficiência na distribuição de modelos durante a execução da transformação. No entanto, não fica claro o quanto as partições derivadas dessa técnica são balanceadas. Além disso, a heurística pode apresentar falso positivo em relação aos elementos do modelo pertencer ou não a uma determinada partição, comprometendo o balanceamento das partições. Os experimentos desse trabalho demonstram que há bons resultados referentes a escalabilidade no processamento de transformação de modelos, mesmo com a execução realizada em três passos: distribuição de dados, transformação

[1]Nesta tese convencionou-se que o particionamento é uma operação que divide o modelo de entrada em tempo de execução. E a fragmentação decompõem o modelo de entrada em fragmentos antes da execução.

paralela local e composição local de modelos, seguindo o fluxo das funções *Map* e *Reduce*.

2. Aracil e Ruiz (2017) propõem a linguagem CloudTL para processar transformação distribuída de modelos em formato Ecore. Os modelos nesse formato são distribuídos tendo como base o formato JSON, RESTfull e aplicações WEB (Aracil e Ruiz, 2016). Os elementos em XMI (modelos e metamodelos) são extraídos para o formato JSON e atribuídos para URLs (*Uniform Resource Locator*). Os elementos multi-valorados são organizados em estruturas de *arrays* em JSON. As regras de transformação escritas em CloudTL são processadas em ambiente cliente-servidor usando o Apache Storm (Apache, 2019a). Para garantir a consistência da transformação, uma aplicação REST (*Representational State Transfer*) recupera os elementos que estão no servidor e os envia para o cliente para tratar a dependência entre os elementos e transformá-los. O modelo é fragmentado de acordo com a quantidade de URLs, cujo fragmentos que possuem elementos muti-valorados são distribuídos pelo servidor aos clientes para o processamento, enquanto que os demais elementos são processados no servidor. A distribuição (clientes) e a junção (servidor) dos elementos para concluir a transformação gera *overheads* na rede entre o servidor e os clientes. Embora a CloudTL tenha apresentado um desempenho superior à ATL, no que diz respeito ao processamento de regras de transformação, a CloudTL não possui mecanismo de *lazy evaluation* e tem restrições em processar strings e inteiros;

3.1.2 TP/DM sem Particionamento ou Fragmentação do Modelo de Entrada

3. Benelallam et al. (2015) propõem uma semântica para a execução distribuída de regras em ATL utilizando o framework MapReduce. Esse trabalho está relacionado ao trabalho #1 no que diz respeito ao alinhamento entre a ATL e o MapReduce, de modo que regras de transformação em ATL possam ser tratadas pela abstração do MapReduce, incorporando o paralelismo implícito à ATL. O particionamento do modelo de entrada é realizado implicitamente pelo MapReduce de acordo com número de nós que executam a fase `Map`. Nessa fase, as regras *match* em ATL são executadas em dois passos: um passo *match* e outro *apply*. Esses passos são respectivamente responsáveis por estabelecer links de rastreabilidade entre os itens de transformação (regras, *matches* e elementos de saída) e pela inicialização dos elementos do modelo de saída. Além disso, a dependência entre os elementos do modelo de entrada é resolvida parcialmente na fase `Map`, em que os elementos interdependentes são assinalados e a resolução da dependência é completada na fase `Reduce`. A fase `Reduce` recebe os dados da fase anterior (fase `Map` finalizada) processa os *trace links* ainda não resolvidos e recompõem o modelo após a distribuição. A computação na fase `Reduce` é realizada em um único nó e não paralelizável. Os experimentos demonstram que o processamento distribuído é escalável. Nesse trabalho, a paralelização e a distribuição são utilizadas somente na fase `Map`, de modo que a primeira fase da transformação é distribuída e a segunda é local. As regras em ATL especificadas nesse trabalho suportam somente o padrão EMF e a serialização em formato XMI. Além disso, os modelos de entrada são carregados por completo na memória principal;

4. Burgueño et al. (2016) utilizam o framework Linda e implementam a abordagem Lintra para o processamento paralelo de transformação de modelos. Essa abordagem é baseada nos trabalhos (Burgueño et al., 2015; Burgueno, 2013; Burgueño et al., 2013), em

que o framework Linda (Gelernter, 1985) (*coordination language* para arquiteturas com *shared-memory*) é empregado para a transformação paralela de modelos. Nessa abordagem, a linguagem Java é usada para as especificações e execuções paralelas de regras de transformação e um conector é disponibilizado para instanciar modelos de entrada e de saída em bancos de dados NoSQL. O paralelismo adotado na abordagem é de responsabilidade do framework Linda (arquitetura *Blackboard* ou *tuple space*). Para validar a abordagem, os autores produzem um *benchmark*, envolvendo a execução de regras de transformação escritas nas linguagens Java (Lintra), QVT, ATL e ETL. O resultado mostra que a abordagem Lintra apresenta um desempenho melhor nas execuções das regras de transformações em relação as outras linguagens utilizadas nos experimentos. No entanto, a abordagem requer que o modelo de entrada seja carregado por completo em memória para ser processado, exigindo disponibilidade de memória em relação ao tamanho do modelo a ser transformado. Além disso, nenhuma estratégia de fragmentação do modelo de entrada é adotada para a abordagem Lintra;

5. GPUs (*Graphics Processing Unit*) têm sido usadas em processamento paralelo, abrangendo domínios de aplicações como: imagens, áudio, bioinformática, entre outros. Nesta perspectiva, Fekete e Mezei (2016) desenvolveram uma ferramenta para transformação de modelos usando GPUs, denominada de GEMP, *GPGPU-based Engine for Model Processing* (GPGPU - *General-purpose Computing on Graphics Processing Units*). A transformação de modelos é baseada em grafos, exigindo que o modelo de entrada seja representado em um grafo e instanciado em uma tabela *hash*, onde o ID de cada vértice e uma lista de seus vizinhos são armazenados. O modelo de entrada é dividido de maneira aleatória e processado em várias rodadas, uma vez que o *matching* dos elementos do modelo é processado em dois passos: *topological match* e *attribute checking*. No primeiro passo, o modelo é mapeado configurando em uma indexação do grafo com elementos numéricos (IDs). No segundo passo, os atributos do modelo são serializados em strings. O resultado do mapeamento consiste em assinalar os elementos do modelo de entrada em um conjunto de *arrays* bidimensionais compatíveis com o Framework OpenCL (Khronos, 2019). O modelo de programação paralela CUDA (*Compute Unified Device Architecure*) favorece o desenvolvimento de aplicações paralelas para GPUs, mas exige do(a) desenvolvedor(a), o controle de alocação de memória e o uso de *buffers*. A abordagem GEMP exige que o modelo de entrada esteja em formato texto e mapeado para um conjunto de grafos que é traduzido para o contexto OpenCL. O particionamento de modelos é realizado de maneira aleatória e não há balanceamento de carga ou estratégia de uniformização dos submodelos (subgrafos). Na GEMP, não há nenhum comparativo de desempenho em relação ao processamento de transformação de modelos, apenas é mencionado que há escalabilidade e um ganho de 10% de desempenho;

6. Benelallam et al. (2018) exploram o modelo de programação MapReduce e o framework Hadoop para o processamento paralelo e distribuído de transformação de modelos. Nesse trabalho um conjunto mínimo das propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) é considerado durante a TP/DM. A inserção de propriedades ACID pode ser considerada uma evolução à abordagens apresentadas nos Trabalhos #1 e #3 apresentados anteriormente. Os mecanismos de tolerância à falhas presentes no framework Hadoop e no modelo MapReduce são explorados em conjunto com as propriedades ACID, visando a consistência da transformação distribuída de modelos. Nessa abordagem a linguagem ATL é inserida para a especificação das regras de transformação e a distribuição e o paralelismo são realizados pelo próprio

ambiente de modo implícito. A abordagem desse trabalho apresenta escalabilidade no processamento de transformação de grandes modelos (100.000 elementos, 3̃2MB), segundo os autores, ainda cabem melhorias como: diminuir o número de pares de chave valor (*key, value*) que são transmitidos entre as fases *Local Match-Apply* e a *Global Resolve*, impactando diretamente no tempo de execução da fase *shuffling* do MapReduce; considerar a compatibilidade da abordagem para os recentes frameworks de processamento distribuído; entre outras melhorias. Tais melhorias são relevantes em relação à escalabilidade e busca a evolução da abordagem proposta. Porém, uma estratégia de fragmentação de modelos, buscando a uniformidade da distribuição e a localização de dados (partições de modelos) no ambiente distribuído são aspectos que podem aprimorar a abordagem;

7. a transformação paralela de modelos é proposta por Tisi et al. (2013), em que o paralelismo implícito para a ATL é implementado e a computação da transformação paralela é direcionada para as regras. A computação é agrupada por regras (*task parallelism*), de modo que cada tarefa executa uma regra diferente (incluindo as expressões em OCL). A abordagem exige que o modelo de entrada esteja totalmente disponível em memória. A execução das regras ainda é dividida em duas fases: *matching* e *application*. Na fase *matching*, todos os *matching* são executados para a criação dos elementos de saída e dos links entre eles. A segunda fase é ativada para completar a transformação somente depois que todos os *matching* tenham sido executados. Isto implica em um ponto de sincronização o que pode afetar negativamente o tempo de execução da transformação (quando uma tarefa não é completada em tempo hábil). Essa abordagem usa somente o paralelismo de tarefas. Para adicionar o paralelismo implícito à ATL, o compilador e a máquina virtual da linguagem ATL foram alterados com a introdução de interbloqueios (*locks*) para o acesso a dados. A abordagem apresenta ganhos de desempenho na transformação de VLMs quando comparada às linguagens ATL (padrão) e Java. A abordagem provê a execução de tarefas em paralelo somente na fase *matching* de transformação e nenhum outro mecanismo como particionamento de modelos ou paralelismo de dados são adotados nessa abordagem;

8. O framework VMTS (*Visual Modeling and Transformation System*) é baseado em grafo e utilizado na metamodelagem com suporte à transformação de modelos. Imre e Mezei (2012) adaptam o VMTS em uma abordagem de transformação paralela de modelos. A adaptação é provida por um algoritmo que executa a transformação em duas fases: a fase *multithreaded matching* usada para encontrar e agregar todos os *matching* sem sobreposição de elementos do modelo de entrada e a fase *modifier*, que executa sequencialmente as modificações em uma única *thread*, formando um ciclo que se repete até que todos os *matches* são processados. Isso significa que as regras são executadas exaustivamente. O VMTS é desenvolvido na linguagem .Net com a modificação no código fonte para refletir o uso adequado de *threads* durante a execução de transformação de modelos. O paralelismo é implementado apenas em uma das fases da abordagem. Mesmo assim, o desempenho em relação ao tempo de execução das transformações em paralelo é menor do que quando comparado com o tempo de execuções em uma única *thread*. No entanto, a abordagem está limitada à quantidade de *matching* a ser processada na primeira fase de execução (forma exaustiva como estratégia). Além disso, na segunda fase a execução do algoritmo é sequencial. Esses aspectos tem influência direta no desempenho da abordagem;

9. Gremlin-ATL é uma abordagem para transformação de modelos desenvolvida com as linguagens Gremlin e ATL, proposta por Daniel et al. (2017). Introduzida no framework Apache TinkerPop, Gremlin (Apache, 2019e) é uma linguagem orientada a fluxo de dados para o processamento de grafos. Expressões em Gremlin podem ser escritas de maneira imperativa, declarativa ou em ambas. Para executar as transformações em ATL diretamente em banco de dados, as construções em ATL são traduzidas para Gremlin, gerando um *script*. Esse *script* é enviado junto com a implementação do modelo de mapeamento e com os *helpers* de transformação à API de execução do framework TinkerPop. A transformação é executada diretamente no banco de dados em que está instanciado o modelo de entrada, isso significa que os modelos em formato XMI são instanciados para o framework NeoEMF/Graph (Daniel et al., 2016) antes da execução da transformação. Os resultados das execuções de transformações de modelos entre a ATL e a ATL-Gremlin mostram que a abordagem proposta apresenta uma melhora significativa no desempenho em termos de tempo de execução e uso de memória. Contudo, essa abordagem requer a tradução da ATL para a linguagem Gremlin e exige que os modelos de entrada estejam armazenados em banco de dados NoSQL compatíveis com o framework NeoEMF/Graph. A ATL-Gremlin não utiliza explicitamente um framework para execução paralela ou distribuída de TM, mas usa frameworks de persistência escalável como o NeoEMF/Graph para executar as transformações de VLMs.

3.1.3 Particionamento/Fragmentação de Modelos

10. Benelallam et al. (2016) descrevem uma abordagem de particionamento de modelos introduzida no Trabalho #1. Essa abordagem é baseada na formalização de particionamento de grafos. As partições geradas pela abordagem são distribuídas entre nós durante a transformação distribuída de modelos. A formalização leva em consideração a maximização de sobreposição de *task-data*, tendo como base um grafo de dependência. Dado um *cluster* de nós (máquinas), o propósito é balancear a quantidade de elementos que são carregados em cada nó. Inclui nessa abordagem, um algoritmo para extração de *footprints* das regras de transformação por meio de uma análise estática. Os *footprints* representam uma abstração do que cada regra de transformação requer para a sua execução, além de compor um grafo de dependência. No entanto, a distribuição de dados requer que o modelo de entrada seja trespassado de acordo com os *footprints*. Para minimizar essa questão, um algoritmo do tipo *greedy* é implementado para particionar e distribuir modelos de entrada sobre um *cluster* de nós. Além disso, um modelo para identificar a dependência entre os elementos do modelo sob o particionamento é usado. O algoritmo de distribuição leva em conta a existência de um framework para a persistência distribuída de modelos. Os experimentos da abordagem mostram que o particionamento aumenta eficiência da transformação de modelos em 16% quando comparada com o particionamento aleatório sem o uso do algoritmo proposto. A abordagem pode alavancar pesquisas no âmbito de particionamento de modelos para transformação distribuída. No entanto, essa abordagem não busca o balanceamento da distribuição do modelo como um todo, ela usa somente os vértices do grafo de dependência para o balanceamento e requer a adoção de um *backend* distribuído. Segundo os autores, o algoritmo utilizado na distribuição está suscetível a falsos positivos podendo desbalancear as partições geradas;

11. a transformação de modelos baseada em *streaming* considera que o modelo de entrada não está na memória principal por completo ou indisponível até o início da transformação. O modelo é inserido de forma contínua no mecanismo de transformação e processado incrementalmente. Cuadrado e de Lara (2013) propõem uma abordagem para transformação de modelos incluindo: mecanismos para especificar fragmentos de modelos, um repositório de modelos e um agendador para executar a transformação de VLMs. Uma unidade de *streaming* é um fragmento do modelo que contém um ou mais elementos e que pode ter referencias do tipo intra-fragmento (entre elementos *containment* e *non-containment*) e ou inter-fragmento (somente entre elementos *non-containment*). Essas características são especificadas em uma DSL, a IDC (*Intermediate Dependency Code*) de baixo nível e compilada para Java. Além da IDC, inclui na abordagem o uso de uma ferramenta para a transformação de modelos denominada Eclectic. O repositório Morsa (de Murcia, 2013) é utilizado nos experimentos que envolvem transformações de modelos grandes. Os resultados dos experimentos mostram que há uma melhora de desempenho em torno de 10% à 15%, mas não é descrito como essas porcentagens são obtidas. A abordagem não suporta transformação paralela ou distribuída de modelos, embora permita fragmentar os modelos de entrada. O uso de *streaming* e a fragmentação de modelos podem favorecer a escalabilidade durante o processamento de transformação de VLMs, uma vez que cada fragmento do modelo pode ser processado em paralelo (memória compartilhada) ou assinalado a um nó (máquina) do sistema distribuído.

12. baseado em conceitos de modularidade e separação de interesses, Amálio et al. (2015) apresentam a FRAGMENTA, uma abordagem modular desenvolvida para a fragmentação e composição de modelos. A abordagem é baseada em descrições algébricas de modelos, teoria de grafos e apresenta um formalismo para o particionamento de modelos sob as perspectivas: *top-down* e *bottom-up*. A modularização é formada pelas primitivas: *fragments*, *clusters* e *models*. A primitiva *fragments* fornece as referências entre os vértices por meio de arestas de referência constituindo os *proxies*, os quais permitem referenciar vértices de outros fragmentos. Já a primitiva *clusters* é usada para a junção de fragmentos relacionados e fornece meios para uma organização hierárquica, em que um *cluster* pode ter outros *clusters* e fragmentos. A primitiva *models* organiza em *clusters* as coleções de fragmentos, os quais podem ser instanciados em pasta de arquivos e os fragmentos em arquivos. A abordagem inclui um modelo de composição de fragmentos como um processo de substituição para a resolução de referencia, em que *proxies* e vértices referenciados são compatibilizados, enquanto que as arestas de referência são eliminadas. No entanto, a fragmentação de modelos é mantida quando há a necessidade de junção dos fragmentos (propósito geral), a qual é baseada na união de todos os fragmentos sem a resolução de referência em uma operação de junção de fragmentos. A abordagem é especificada em linguagem Z[2], cujas expressões são analisadas e validadas em Isabelle[3]. A abordagem não é experimentada com grandes modelos, mas possui um formalismo que pode direcionar novas pesquisas e evoluir a ideia de organizar fragmentos de modelos em *clusters* considerando a possibilidade de distribuir esses *clusters* em nós de máquinas;

13. Scheidgen et al. (2012) propõem uma abordagem para dividir modelos em fragmentos para facilitar a manipulação de VLMs em operações de modelagem. A abordagem utiliza

[2]https://cse.buffalo.edu/LRG/CSE705/Papers/Z-Ref-Manual.pdf
[3]https://isabelle.in.tum.de/

um metamodelo de fragmentação que determina os recursos que geram referências do tipo *containment* e também *cross-references*. As referências *containment* são marcadas de maneira que a hierarquia com essas referências sejam fragmentadas, formando as referências inter-fragmentos. Somente as referências *containment* determinam a fragmentação, mas as referências do tipo *cross-references* podem ser referências inter-fragmentos por acidente. Nessa abordagem inclui a utilização do framework EMFFrag[4] e a seguinte estratégia de fragmentação: o modelo é fragmentado de acordo com a sua hierarquia de referência *containment* e os fragmentos gerados são disjuntos de modo que nenhum objeto fará parte de dois fragmentos. As referências entre fragmentos são do tipo inter-fragmentos e as referências dentro de um fragmento são do tipo intra-fragmentos. IDs são usados nas referências inter-fragmentos para identificar objetos na hierarquia *containment* (similar a uma expressão XPath[5]), como também permitir junções dos fragmentos (pós fragmentação). A abordagem é comparada com os frameworks de persistência de modelos como EMF, CDO e Morsa, de acordo com o resultado da comparação há ganhos significativos de desempenho. Os autores afirmam que não há um tamanho ideal para cada fragmento, mas que tamanhos intermediários fornecem uma boa aproximação em relação ao tamanho versus desempenho. Embora os autores não apresentam quantitativamente o que é um tamanho intermediário, eles fazem as seguintes suposições: considera-se uma fragmentação somente quando os fragmentos têm o mesmo tamanho e que qualquer fragmentação é considerada ótima em relação à operação *partial-load*. O resultado da abordagem não foi utilizada para transformação de modelos, mas ela pode servir como base para fragmentação de modelos de entrada em abordagens para TP/DM.

14. nos cenários em que o modelo de entrada é totalmente carregado para a memória principal, o consumo de espaços de memória é proporcional e linear ao tamanho desse modelo. Para esses cenários, Wei et al. (2016) sugerem uma abordagem em que o acesso *read-only* ao modelo é suficiente e que parte(s) de interesse do modelo (*EObjects*) sejam carregadas na memória, ignorando o restante dos elementos do modelo. Para isso, os autores criaram a ferramenta SmartSAX como uma extensão para o framework EMF capaz de particionar o carregamento de modelos em XMI para a memória principal. O acesso às partições em memória é transparente. No entanto, o particionamento apresenta algumas limitações como: somente a utilização de elementos em referências *non-containment* para obter os IDs (indexação dos elementos) que não dependam das suas posições na hierarquia *containment*. Isto significa que os elementos em referências auto-contidas não são particionados; não permite que as mudanças feitas nos fragmentos do modelo carregado em memória se propaguem de volta para o modelo de entrada; não sustenta o carregamento de modelos que estão instanciados em múltiplos arquivos XMI. Embora modelos em formato XMI sejam amplamente usados, VLMs em XMI quando são submetidos à operações do tipo *read-write*, apresentam alto consumo de memória, afetando o desempenho do processamento de tais operações. Para esses casos, a abordagem sugere o formato JSON como uma alternativa em relação ao formato XMI para expressar modelos;

15. os conceitos de modularização (projeto, pacote e unidade) e anotações em elementos de metamodelos são aplicados por Garmendia et al. (2014) em uma abordagem que permite a edição de modelos de acordo com uma estrutura modular. A abordagem

[4]https://github.com/markus1978/emf-fragments

[5]Uma expressão XPath pode ser usada para navegar e selecionar nós de um documento em XML

também provê a composição de partes do modelo para sua reconstrução. Para permitir o uso da abordagem os autores desenvolveram a ferramenta EMF-Splitter (especificada em Acceleo [6]), uma linguagem para geração de código baseada em MOF (*MetaObject Facility*). Essa ferramenta é um *plugin* para o EMF (Eclipse, 2019c) que define uma estruturação de modelos, visando a criação de modelos de modo colaborativo. Os conceitos usados como base para o particionamento e também para a composição das partes do modelo são aspectos relevantes para uma abordagem direcionada ao particionamento de modelos. No entanto, a abordagem permite o particionamento de modelos somente em formato XMI. Nesse trabalho não há experimentos de particionamento de grandes modelos. Além disso, não está descrito como as interconexões entre os elementos do modelo são tratadas pela abordagem;

16. Fan et al. (2020) demonstram por meio de um formalismo que o particionamento de grafos é um problema NP-completo (Aaronson, 2005) (NP *Non-deterministic Polynomial*), uma vez que grafos podem ser extensos e ter atualizações, dificultando a manutenção das partições. Os autores propuseram uma abordagem baseada em particionamento incremental a fim de minimizar a complexidade em relação ao particionamento de grafos que são atualizados. Incluem na abordagem a formalização de particionamento de grafos e um conjunto incrementadores que integrados aos particionadores permitem o particionamento de vértices e arestas. Os incrementadores são delimitados por heurísticas e implementados em C++ com a biblioteca OpenMP[7] e o pacote ParMETIS[8]. A validação da abordagem utiliza um conjunto de grafos extraídos de aplicações como redes sociais e sistemas de *hyperlinks*, como também grafos gerados em ambientes controlados (250 milhões vértices e 6 bilhões de arestas). Um conjunto de particionadores como FENNEL, KGGGP, entre outros são integrados aos incrementadores da abordagem (e.g., incFENNEL, incKGGGP), ...). Os particionadores desse conjunto são comparados com os particionadores que possuem os incrementos da abordagem (e.g., FENNEL vs incFENNEL). De acordo com os experimentos, a abordagem é promissora sob as seguintes perspectivas: eficiência em uso computacional, escalável e particionamento balanceado. No entanto, a abordagem limita-se em oferecer incrementadores a um conjunto de algoritmos particionadores, mesmo assim a ideia de particionadores incrementais pode favorecer o particionamento de modelos;

17. no trabalho de Clasen et al. (2012) não há a especificação ou implementação de transformação distribuída ou paralela de modelos, e tampouco contempla o particionamento de modelos. Esse trabalho, foi selecionado e classificado porque apresenta um roteiro de pesquisa sobre a transformação de modelos baseada em computação em nuvem. O roteiro é discutido sob duas perspectivas: armazenamento e transformação de modelos. Os autores apresentam questões de pesquisa que envolvem modelo de armazenamento, ferramentas de acesso aos elementos do modelo e principalmente a dificuldade de distribuir os elementos do modelo e seus relacionamentos. Nesse trabalho, são abordados aspectos relacionados ao processamento paralelo de transformação de modelos, como também a composição de modelos após esse processamento. Também são descritas e discutidas alternativas para o processamento paralelo/distribuído de transformação de modelos. Embora não haja exemplos práticos a respeito da transformação paralela

[6]https://www.eclipse.org/acceleo/
[7]https://www.openmp.org/specifications/
[8]http://glaros.dtc.umn.edu/gkhome/

ou distribuída de modelos, esse trabalho tem ajudado no planejamento de recentes pesquisas, envolvendo a TP/DM.

Na Tabela 3.2 são apresentados os demais itens da classificação de trabalhos, como o item Abordagem, adotado de acordo com as considerações de Kahani et al. (2019), em que a abordagem pode ser: Relacional; Imperativa; baseada em Grafo e Híbrida (combinação de mais de um tipo). As especificações de transformação de modelos que usam a Abordagem Relacional, concentram-se nos elementos de entrada que são transformados em correspondentes elementos de saída. A Abordagem Relacional é baseada na definição de relacionamentos entre os elementos dos modelos de entrada e de saída. Transformação de modelos baseadas em grafo, consiste de um conjunto de regras de transformação de grafo (rules *rewriting* ou regras de produção) aplicado ao grafo de entrada para produzir um grafo de saída. Cada regra consiste de um grafo LHS (*Left-Hand Side*), um grafo RHS (*Right-Hand Side*) e um grafo opcional de condição negativa. A LHS especifica um modelo de subgrafo para o qual a regra pode ser aplicada, enquanto que a RHS especifica o novo modelo de subgrafo correspondente.

O resultado da classificação dos 17 trabalhos que compõem o primeiro grupo, mostra que 9 trabalhos (#1 à #9) utilizam uma abordagem para transformação de modelos. A abordagem Relacional está presente em 6 dos 9 trabalhos, enquanto que a Abordagem baseada em Grafo é usada em 3 trabalhos. Em contra partida, os trabalhos classificados na posição #1, #2 e #10 adotam grafos do tipo direcionado para apoiar suas estratégias de particionamento de modelos. Quanto ao item Framework (terceira coluna da Tabela 3.2), observa-se que há uma miscelânea de tecnologias experimentadas, incluindo a adaptação da ATL (#7) para TP/DM utilizando frameworks como o MapReduce. Essa adaptação provê o paralelismo implícito e é utilizado em 17% (#1,#3 e #6) do total de 17 trabalhos. Além do MapReduce, os frameworks Linda, Apache Storm, TinkerPop e Parallel ATL (#2,#3,#7,#8) também oferecem mecanismos para a abstração do paralelismo.

O item Linguagem de programação permite identificar quais as linguagens utilizadas no universo desta pesquisa. Do grupo de 17 trabalhos selecionados, 35% desse grupo utiliza a linguagem ATL. Ao considerar somente os trabalhos relacionados à transformação de modelos, o uso da ATL é de 64%. Esse resultado corrobora com os resultados obtidos no *survey* de Kahani et al. (2019), em que a ATL está presente em várias facetas e aspectos como uma linguagem para especificar regras de transformação ou como uma semântica declarativa para modelo de transformação. A linguagem Java está presente em 4 dos trabalhos classificados (23%).

Uma vez que as linguagens são identificadas nos trabalhos selecionados, o Estilo utilizado nas implementações indica como as regras de transformação ou as instruções de particionamento/fragmentação são especificadas. O estilo declarativo é usado em 7 dos 17 trabalhos (41%), seguido pelo estilo imperativo com 6 trabalhos (35%). Linguagens que implementam transformação em estilo declarativo funcional ou lógico. Uma linguagem de transformação lógica possui recursos adequados para abordagens relacionais, tais como busca, propagação de restrições e *backtracking*. Em linguagens funcionais, a transformação é descrita como a composição de um conjunto de funções que mapeiam os elementos do modelo de entrada para os elementos correspondentes de saída. Transformações especificadas em linguagens imperativas são como uma lista de ações ou regras executadas sequencialmente. Os conceitos e os construtores de uma linguagem de transformação imperativa são similares às linguagens imperativas de propósito geral (Kahani et al., 2019).

Os Trabalhos #1 e #2 implementam a TP/DM e aplicam uma estratégia de particionamento de modelos explícita (PM). Por exemplo, no Trabalho #1 é adotado

uma estratégia própria de particionamento de modelos não interferindo no particionamento implícito do modelo MapReduce. O particionamento em tempo de execução ou a fragmentação de modelos tem sido usado para vários propósitos, tais como edição, transformação, manipulação colaborativa de modelos, entre outros. Dos 17 trabalhos classificados, 6 deles apresentam particionamento ou fragmentação de modelos, sendo que os Trabalhos #9 e #10 empregam o particionamento de modelos para transformação. Os trabalhos #11 à #15 implementam a fragmentação de modelos de propósito geral. O particionamento de grafos é abordado no Trabalho #16, por meio de uma estratégia que utiliza incrementos adicionados a um conjunto de algoritmos particionadores de grafos. A ideia parece promissora, mas não é dedicada ao particionamento/fragmentação de modelos (PM).

Tabela 3.2: Classificação de Trabalhos Relacionados à TP/DM com ou sem PM (parte 2)

#	Abordagem	Framework	Linguagem	Estilo	Ano
1	Relacional	MapReduce	ATL	Declarativo	2016
2	Relacional	Apache Storm	ATL	Declarativo	2017
3	Relacional	MapReduce	ATL	Declarativo	2015
4	Relacional	Linda	Java	Imperativo	2016
5	Grafo	OpenCL	C++	Imperativo	2016
6	Relacional	MapReduce	ATL	Declarativo	2018
7	Relacional	Parallel ATL	ATL	Declarativo	2013
8	Grafo	VMTS	.Net	Imperativo	2012
9	Grafo	TinkerPop	ATL/Gremlin	Declarativo	2017
10	-	Apache HBase	Java	Imperativo	2016
11	-	Morsa	IDC/Eclectic	Híbrido	2013
12	-	Isabelle	linguagem Z	Formal	2015
13	-	EMFFrag	Java	Imperativo	2012
14	-	EMF	Java	Imperativo	2016
15	-	EMF	Acceleo	Declarativo	2014
16	incremental	OpenMP	C++ e ParMETIS	Imperativo	2020
17	-	-	-	-	2012

O resultado da classificação dos trabalhos relacionados à TP/DM mostra que o paralelismo implícito é o meio utilizado para o processamento dessas transformações, principalmente com a junção da linguagem ATL aos frameworks que fornecem o paralelismo implícito. Na maioria dos casos (7 dos 9 trabalhos), o particionamento de modelos e o paralelismo de tarefas ficam sob a responsabilidade dos frameworks adotados nas execuções das transformações, sem nenhum tipo de interferência. A opção de não interferir no paralelismo implícito do ambiente de execução pode não ser uma estratégia atrativa em termos de desempenho de processamento, visto que as características do modelo de entrada (e.g., tamanho, complexidade estrutural) e o domínio de transformação podem impactar no desempenho do processamento paralelo/distribuído. Do grupo de 17 trabalhos classificados, apenas os trabalhos (dois) (Aracil e Ruiz, 2017) e (Benelallam, 2016) implementam estratégias de particionamento, além do particionamento implícito do ambiente de execução. Isso significa que ainda há espaços para a fragmentação/particionamento de modelos no processo de transformação paralela/distribuída de modelos. A estratégia de correlacionar a fragmentação de modelos de entrada ao particionamento de dados do ambiente de execução, como também interferir nesse particionamento podem contribuir com o desempenho desse processo. De acordo com a classificação dos trabalhos contidos nas Tabelas 3.1 e 3.2, tal estratégia ainda não foi experimentada no processo de transformação de modelos. Na próxima seção, o segundo grupo de trabalhos é apresentado com a descrição e a classificação de trabalhos relacionados à Dc.

3.2 TRABALHOS RELACIONADOS À ABORDAGEM ORIENTADA A DADOS

A seleção de trabalhos relacionados à Dc segue a metodologia descrita na seção anterior, assim como as bases de dados de busca. Porém, há uma nova lista de *strings* de busca (Listagem 3.2), a qual é utilizada nas buscas de trabalhos.

Listagem 3.2: *String* de Busca para Dc

```
(((model-driven engineering) OR (MDE) OR (model driven engineering) OR (model transformations)
   OR (transformation rules)) AND ((Data-centric) OR (data centric) OR (declarative style)));
   ((Data-centric) OR (data centric))
```

A seleção de trabalhos contidos nos resultados das buscas segue os critérios de inclusão e de exclusão. Os critérios de inclusão estão definidos como:

- trabalhos que utilizam a Dc como uma abordagem aplicada na solução de problemas de propósito geral, sobretudo em operações relacionadas à MDE;
- trabalhos que implementam ou combinam explicitamente os principais aspectos da abordagem Dc como operações sobre dados, modelos de dados, paradigmas de implementação de alto nível (e.g., declarativo) e paralelismo implícito empregados para propósito geral.

São critérios de exclusão dos resultados das buscas, a não aplicação dos aspectos da Dc ou a não explicitação do uso da Dc em aplicações de propósito geral.

A definição dos critérios de inclusão e exclusão está pautada nos principais aspectos de uma abordagem orientada a dados (Zhou e Demsky, 2010; Alvaro et al., 2010; Aki-Hiro, 2014; Zu et al., 2016), cuja prioridade são operações sobre dados em alto nível de maneira escalável. Na abordagem Dc, considera-se que a distribuição e o paralelismo devem ser transparentes ao desenvolvedor(a) sem a necessidade de atentar-se sobre sincronismo, *scheduling*, comunicação e outros controles utilizados na movimentação de programas e dados entre os nós de um ambiente paralelo/distribuído. Assim como no primeiro grupo, os trabalhos são classificados em uma numeração decrescente de importância conforme os critérios de classificação. Os critérios estão organizados nas colunas da Tabela 3.3, exceto na coluna # que contem a indicação de classificação de cada trabalho e a coluna Trabalho que possui uma referência do trabalho classificado.

O critério de classificação Domínio de Problema é utilizado de modo que se possa identificar o domínio do problema abordado pelo trabalho, e quais aspectos da Dc são aplicados. O critério MDE é para distinguir os trabalhos que empregam a Dc em operações relacionadas à MDE, neste caso os trabalhos que atendem a este critério têm prioridade na classificação (Tabela 3.3), em que a coluna MDE é sinalizada com "$\surd$". Por fim, o critério Tecnologias indica as tecnologias utilizadas nos trabalhos classificados. Os trabalhos classificados são descritos e em seguida organizados na Tabela 3.3:

1. Batory et al. (2013) reinterpretam a MDE sob o ponto de vista de Banco de Dados Relacional. Os autores têm como propósito simplificar o entendimento dos conceitos da MDE, utilizando uma abordagem orientada a dados e uma linguagem declarativa para a transformação de modelos (M2M). Para isso, as regras de transformação são mapeadas para a linguagem Prolog[9], permitindo a especificação de regras declarativas para as transformações de modelos, assim como o mapeamento de metamodelos para

[9]Prolog é uma linguagem de propósito geral para programação baseada em lógica matemática (Diaz, 1999)

tabelas relacionais. A abordagem foi proposta como um método para explicar a MDE para estudantes de graduação. Os autores consideram que a MDE pode ser entendida como uma aplicação de Banco de Dados Relacional. A metodologia proposta para explicar a MDE incluem as transformações M2T (*Mdel2Text*) e T2M (*Text2Model*). Nas transformações M2T, o template Apache Velocity (Apache, 2019b) é modificado para integrar com as bases de dados em Prolog. Nas transformações T2M, um parser é escrito em Java para converter modelos em XML para tabelas em Prolog. As regras dessas transformações são especificadas na linguagem Java. Do ponto de vista acadêmico, Batory e Azanza (2017) complementam o trabalho anterior com estudos de casos envolvendo estudantes de graduação em computação. De acordo com os resultados, os autores consideram que os estudantes não precisam ter familiaridade com Banco de Dados Relacional para entender a MDE, mas exige o conhecimento em normalização (a abordagem requer tabelas normalizadas) e programação de computadores. Além da aprendizagem em MDE, a abordagem proposta utiliza uma linguagem declarativa orientada a dados e organiza os dados de transformação em tabelas. De acordo com os autores, um importante aspecto da abordagem é o baixo volume de código declarativo escrito em Prolog para especificar as regras de transformação (M2M);

2. Thang et al. (2011) apresentam um conjunto de ferramentas baseado em MDE para apoiar o desenvolvimento de aplicações orientada a dados em rede de sensores. Nesse trabalho, metamodelos e uma DSL são projetados e implementados para a especificação de uma aplicação para processar dados em nós de sensores. Incluem nesse processamento, tarefas como *sampling, aggregation* e *forwarding*. A DSL e as ferramentas apoiam a transformação de modelos gerando código fonte para a aplicação (M2T). Conceitos como a disseminação de mensagens entre os nós de sensores; processamento *in-network* (cada nó sensor tem autonomia para avaliar o conteúdo de uma mensagem antes de tomar uma decisão); operação *in-network* (o nó sensor transfere dados e pode controlar funcionalidades) são observados por metamodelos e pela DSL. A abordagem utiliza os níveis de abstração da MDA (*Model-Driven Architecture*) para especificações de modelos PIM (*Platform Independent Model*) e PSM (*Platform Specific Model*). No nível PIM, o metamodelo e a DSL são utilizados para especificação da aplicação em desenvolvimento. Já no nível PSM, é descrito o processo para transformar as especificações do nível PIM para o modelo de saída no nível PSM. A transformação provida nesse trabalho é de M2T, em que o código para a linguagem *nesC*[10] é gerado dos modelos de entrada. Para isso, plataformas como a TinyOS[11] são usadas como plataformas de destino e o mapeamento entre os modelos PIM e PSM são especificados em ATL. Nesse trabalho, um modelo de desenvolvimento de software dirigido por modelos e baseado na Dc é proposto para o desenvolvimento de aplicações em rede de sensores;

3. embasados nos paradigmas de programação lógica e programação declarativa, Almendros-Jiménez et al. (2016) utilizam a PTL (*Prolog based Transformation Language*) para especificar regras de transformação (Almendros-Jiménez e Iribarne, 2013), criando um framework para TM. A PTL é uma linguagem que permite a combinação do estilo da linguagem ATL com a programação lógica. Há uma demonstração em que as expressões declarativas em Prolog são úteis para as especificações de regras de TM. Nesse trabalho, os autores levam em conta que um metamodelo é um diagrama de classes, o qual é interpretado como um grafo contendo dois tipos de nós: classes e atributos,

[10]http://tinyprod.net/docs/nesc-ref.pdf
[11]http://www.tinyos.net/

e dois tipos de arestas: atributos membros e associações. O grafo é representado em Prolog e as operações sobre ele são especificadas também em Prolog por meio de seus predicados. Os autores definiram uma semântica declarativa direcionada à linguagem PTL para metamodelos e modelos. Para avaliar a abordagem foram especificados mapeamentos entre ATL e PTL, OCL e Prolog de modo que, um subconjunto da PTL é equivalente a um subconjunto declarativo da ATL (PTL como uma implementação da ATL). Essa abordagem possui vários pontos relevantes, como por exemplo: o uso do estilo declarativo para especificar regras de transformações e restrições de validações e a utilização de triplas em RDF (*Resource Description Framework*) para instanciar modelos. O desempenho da abordagem é penalizado para médios e grandes modelos, principalmente quando há o uso de múltiplos *joins* nas especificações de restrições;

4. Ge et al. (2012) propuseram uma abordagem para o desenvolvimento de modelos executáveis em arquiteturas SoS (*System-of-Systems*[12]). Essa abordagem é composta pela junção do DM2 (*Meta-model*) do DoDAF (*Department of Defense Architecture Framework*) e da Dc. Essa abordagem visa flexibilidade e adaptabilidade na construção de modelos executáveis. A abordagem proposta é orientada por um conjunto de processos, tais como: um processo para estabelecer metamodelos de dados arquiteturais e metamodelos executáveis; um formalismo para guiar a modelagem de dados arquiteturais e definir modelos executáveis; um processo para definir as regras de mapeamento entre metamodelos e um processo para realizar a transformação de modelos de acordo com as regras de mapeamento para o formato XML. As regras de mapeamento podem ser expressadas em XSL (*eXtensible Stylesheet Language*) para produzir automaticamente um modelo executável. Expressões em XPath (*XML Path Language*) são especificadas para selecionar nós do documento em XML que contêm dados da instância arquitetural necessários para construir os modelos executáveis. Já a XSLT (*XSL Transformation*) pode ser empregada para definir o modelo de transformação aplicado ao documento de entrada em XML, contendo a instância arquitetural para transformar esse documento em modelos executáveis em XML. No entanto, esse trabalho não apresenta uma validação da abordagem, apenas ilustra um procedimento para transformação de uma rede de Petri colorida em um modelo executável;

5. nas séries Springer (DCSA) *Data-centric Systems and Applications* há diversas publicações de trabalhos em diferentes domínios de aplicações, entre elas o trabalho de Bonifati et al. (2011). Nesse trabalho, o mapeamento de esquemas em transformações, alterações e integração de dados são abordados. Esse trabalho aborda como um problema, as diferenças e incompatibilidades entre formatos e modelos heterogêneos de bancos de dados. Especificamente o mapeamento entre os bancos de dados relacional de origem e destino. Os autores discutem as questões que envolvem o desenvolvimento e aplicações de mapeamento de esquemas como: a expressividade do mapeamento de esquemas para cobrir conflitos entre dados e metadados; mapeamento de esquemas para o modelo de dados complexos, como em XML; o uso de mapeamento em larga escala em ambientes distribuídos e a normalização otimizada de mapeamento de esquemas. Esses problemas são discorridos sob os aspectos formal e prático relacionados ao mapeamento de esquemas, incluindo ferramentas que tratam cada problema mais especificamente.

[12]SoS é um conjunto de sistemas ou elementos de um sistema que interagem para fornecer um recurso exclusivo que nenhum dos sistemas constituintes pode realizar sozinho (ISO/IEC/IEEE 21839)

Desse modo, tecnologias como a Datalog[13], XML, *XQuery*[14], XSLT[15], OWL[16], sistemas P2P, entre outras são incorporadas na discussão. No entanto, não há experimentos que corroboram com as discussões apresentadas, além de alguns exemplos em SQL. Os problemas apresentados nesse trabalho são parecidos com questões encontradas em operações de transformação de modelos, por isso considera-se que esse trabalho tem relação com a MDE;

6. o processamento de grafos em larga escala é discutido no trabalho de Shao e Li (2018). Nesse trabalho, são apresentados os desafios e os princípios para o desenvolvimento de sistemas direcionados ao processamento paralelo de grafos (grafos com milhões de arestas e vértices) de maneira escalável. Um conjunto de abordagens e sistemas utilizados no processamento de grafos são analisados e classificados, tais como Neo4j[17], Pregel(Malewicz et al., 2010), Giraph[18], GraphX[19], entre outros. Também são abordados algoritmos para o *matching* de subgrafos, como GraphQL, RDF-3X, R-Join, e outros. Quanto ao processamento de grafos, duas abordagens são tratadas: a) processamento de consultas online e b) análise offline. Na primeira abordagem, duas técnicas para o processamento distribuído de grafos são apresentadas: a *asynchronous fanout search* e a *index-free query processing*. A primeira técnica é discutida sob os aspectos: algoritmos *matching* em subgrafos, indexação do grafo, a influência das operações *join*, particionamento de grafos no desempenho dos algoritmos *matching* e o processamento de grafos com milhões de vértices e arestas em ambientes distribuídos. Quanto à segunda técnica, os autores incluem o MapReduce como uma alternativa para o processamento paralelo/distribuído de grafos, porém o desempenho do processamento é alcançado quando há um modelo eficiente (balanceamento, localidade de dados) para particionamento de grafos, visto que os mecanismos de particionamento do MapReduce é custoso para processamento de grafos. Uma outra alternativa citada é a aplicação do modelo *vertex-centric computation*, em que o framework Pregel explora o paralelismo em nível de vértices e não movimenta partições de grafos sobre a rede. Essa alternativa, utiliza a troca de mensagens entre os vértices a cada iteração durante o processamento reduzindo o tráfego na rede. Além dessas alternativas, questões relacionadas à otimização da comunicação são abordadas para a computação distribuída de grafos, como um *trade-off*, vértices de um grafo em máquina local ou vértices em qualquer máquina remota. Por fim, os autores consideram o uso da técnica *Local Sampling* como uma possível maneira de eliminar o gargalo de comunicação na rede, já que essa técnica busca processar porções de grafos em máquinas locais e em seguida agregar o resultado final de cada máquina em uma única máquina. Nesse trabalho, não há registros de experimentos com as técnicas descritas e ferramentas apresentadas, mas há sugestões para o desenvolvimento de aplicações orientadas ao processamento intensivo de grafos;

7. Laptev et al. (2016) propõem uma abordagem para DSMSs *Data Stream Management Systems*, em que a ESL (*Expressive Stream Language*) é projetada para maximizar o espectro de aplicações que um DSMS pode suportar de maneira compatível com os padrões da linguagem SQL. A ESL aceita consultas em SQL, permite a atualização

[13]https://docs.racket-lang.org/datalog/
[14]https://www.w3.org/TR/xquery/all/
[15]https://www.w3.org/Style/XSL/
[16]https://www.w3.org/OWL/
[17]https://neo4j.com/
[18]https://giraph.apache.org/
[19]https://spark.apache.org/graphx/

de tabelas em banco de dados e consultas contínuas em *stream* de dados de modo declarativo (`CREATE STREAM`). A ESL possibilita a especificação de um *wrapper*, o qual permite a importação de dados com um *timestamp* associado ao *stream*. Os operadores como `STREAM` e `TIMESTAMP` são requeridos em declarações ESL. Em termos de semântica, os operadores da ESL produzem os mesmos resultados quando aplicados aos *stream* de dados ou em tabelas de banco de dados, nas quais novas tuplas são continuamente anexadas. A ESL aceita a condição *Windows*[20] em UDAs (*User-Defined Aggregates*) arbitrários, incluindo *Windows* lógica e física, e *slides* (extensões da SQL em agregações contínuas). As especificações de *windows* são em modo declarativo, por exemplo `CREATE WINDOW AGGREGATE`. Nesse trabalho, o *Data mining stream* é abordado como um desafio ao DSMS e neste caso inclui o algoritmo *Density-Based Clustering (DB-Scan)* para minerar e monitorar *stream* de dados. Nessa abordagem, o conceito de *Drift*[21] é aplicado para modelar e capturar a tendência de tempo e padrões nos *streams* durante o processamento de tais *streams*. O *Stream Mill System* foi projetado em uma arquitetura cliente-servidor para processar a abordagem. O cliente possui uma interface para a submissão de comandos e visualização dos resultados dos comandos submetidos. A abordagem é capaz de processar consultas contínuas de mineração de dados e modelos em XML. No entanto, nesse trabalho não há registros de experimentos demonstrando o desempenho da abordagem, apenas exemplos declarativos de código fonte;

8. Aderholdt et al. (2018) estenderam a biblioteca SharP[22] buscando aprimorá-la para suportar os seguintes aspectos: *data locality, data resilience* e *data movement*. Esses aspectos são discutidos e explorados sob a ótica de desenvolvimento de aplicações científicas. A biblioteca SharP foi desenvolvida para facilitar o gerenciamento de estruturas de dados como *arrays* e *hashes* em memórias heterogêneas e hierárquicas em um *cluster*. Para tanto, os autores implementaram diretivas em MPI *Message Passing Interface* a fim de permitir aos usuários interações à biblioteca SharP por meio de parâmetros. As modificações orientada a dados incorporadas à biblioteca SharP foram experimentadas, utilizando os *benchmarks* QMCPack e Graph500 em um ambiente com 18.688 nós, cada nó equipado com uma GPU de 32 GB e interconectados com *Cray Gemini*. Os resultados dos experimentos mostram ganhos significativos de desempenho, além de prover os aspectos orientado a dados à biblioteca SharP;

9. Liu e Mellor-Crummey (2013) abordam problemas relacionados à localidade de dados em processamento paralelo/distribuído. Os autores propõem uma solução orientada a dados em que as ferramentas de desempenho do HPCToolkit[23] são estendidas com capacidade para medir e analisar a execução de programas em sistemas paralelos escaláveis. Para suportar a análise orientada a dados, as variáveis do programa sob análise são rastreadas sob as perspectivas de alocação e desalocação estática e *heap* de memória. Variáveis que não pertencem à alocação estática ou *heap* são tratadas como dados desconhecidos. A abordagem proposta confere ao HPCToolkit a propriedade de quantificar a localidade de dados temporal, espacial e NUMA (*Non-Uniform Memory*

[20] Windows é uma condição especial especificada para operadores de consulta *stream* com *blocking* e *stateful* (Li, 2008)

[21] Drift refere-se às mudanças nas relações entre os dados de entrada e saída ao longo do tempo (Wang e Abraham, 2015)

[22] SharP é uma biblioteca para gerenciamento de estruturas de dados em *clusters* (Venkata et al., 2017)

[23] http://hpctoolkit.org

Access) em processadores do tipo IBM POWER7, AMD Opteron, Intel Itanium2 e Ivy Bridge. O HTCtoolkit orientado a dados foi avaliado em cinco diferentes *benchmarks* de programas paralelo codificados nas linguagens C, C++ e Fortran, incluindo as bibliotecas MPI e OpenMP. As avaliações envolveram o código e variáveis de tais programas, permitindo identificar e otimizar os gargalos provenientes da localização de dados. Além disso, o desempenho dos *benchmarks* usados nos experimentos foram aumentados entre 13 a 53%, conforme os resultados apresentados;

10. defeitos relacionados à concorrência como *data races* tendem a ser custosos de rastrear e eliminar, quando presentes em programas orientado a objetos com milhares de linha de código. Há abordagens que buscam impedir *data races* protegendo as sequências de instruções com operações de sincronização. Essas abordagens centradas no controle apresentam fragilidades, pois a garantia de que todos os locais de memória acessados simultaneamente sejam protegidos de forma consistente é de responsabilidade do(a) desenvolvedor(a). Com a abordagem orientada a dados para esse contexto, o controle de concorrência está na identificação e localização de conjuntos de memória que compartilham a mesma propriedade de consistência. Esses conjuntos são agrupados para que sejam atualizados de maneira atômica. Isto significa que o(a) desenvolvedor(a) não precisa ter foco no fluxo de controle do programa. Como também, não precisa especificar onde ou qual o tipo de operação de sincronização será usada, uma vez que cada conjunto atômico possui unidades de trabalho que visam preservar a consistência. O código de sincronização é gerado automaticamente pelo compilador, mas com a opção de inserção do tipo de sincronização pelo desenvolvedor(a). Em vista disso, Dolby et al. (2012) avaliam a aplicabilidade e os benefícios da Dc em face a uma linguagem orientada a objetos. Para esse fim, uma extensão da linguagem de programação Java (AJ) foi desenvolvida com as características da abordagem Dc para sincronização. A AJ estende a sintaxe da linguagem Java com anotações necessárias para suportar o modelo de sincronização orientado a dados. Em AJ, um conjunto de anotações podem ser especificadas para as classes de um programa de maneira declarativa (`atomicset`, `atomic()`). Os experimentos apresentados nesse trabalho revelaram uma baixa sobrecarga de anotações, isto significa que em média foram encontradas 40 anotações para cada KLOC de código fonte em Java, uma relação de 0.6 para 11.5 anotações por KLOC para outras abordagens. No entanto, o desempenho da abordagem não foi satisfatório quando comparado com a versão original do Java usando o *benchmark* SPECjbb[24];

11. Kim et al. (2019) utilizam o paradigma *Approximate Computing* (AxC) para minimizar o gargalo de largura de banda de memoria durante a computação intensiva. Uma abordagem orientada a dados é proposta para a AxC que pode ser usada para melhorar o desempenho de softwares em plataformas de computação *off-the-shelf* (sem alterações no hardware). A ideia principal é modularizar os acessos às estruturas de dados de modo que a largura de banda da memória utilizada seja reduzida. Para isto, foi proposta a técnica denominada *Data Subsetting*, em que os acessos às estruturas de dados são restritos a um subconjunto de elementos dessas estruturas. Os acessos limitados à estrutura de dados minimizam o *footprint* de memória, tornando o tráfego de memória mais *cache-friendly* e aumentando o desempenho. Os desafios para a técnica proposta consiste em identificar um subconjunto de elementos representativos à estrutura de

[24] https://www.spec.org/jbb2015/

dados de entrada. Para a seleção desses elementos leva-se em conta o tamanho do subconjunto e o tipo de elemento, uma vez que eles são baseados na aplicação e nas características dos dados. Considera-se também que cada programa tem suas próprias particularidades. Um outro desafio, refere-se à forma de como os acessos aos dados são aproximados. Por exemplo, quando uma aplicação tenta acessar dados que ficam fora do subconjunto escolhido, o acesso precisa ser redirecionado para uma localização de memória que contenha o subconjunto. Para a técnica *Data Subsetting* foi desenvolvido o template `SubsettableTensor`. Por meio desse template, é possível realizar *Data Subsetting* de modo minimamente intrusivo à aplicação de software através de suas extensões para estrutura de dados. A *Data Subsetting* permite especificar no programa o controle de quais elementos da estrutura de dados são partes acessíveis e como acessá-las. O uso de *subset buffer* permite identificar a localidade espacial entre os elementos de dados que são acessíveis, como também eliminar a computação redundante. Um *benchmark* com um conjunto de aplicações de aprendizagem de máquina foram utilizados para avaliar a *Data Subsetting*, em que implementações em C++ são paralelizadas com a biblioteca OpemMP viabilizando o *benchmark*. Os resultados dos experimentos mostram um significativo ganho de desempenho com processamento paralelo ao utilizar a *Data Subsetting*.

De acordo com a Tabela 3.3, constata-se que há uma intersecção entre as abordagens Dc e MDE em cinco dos 11 trabalhos (#1-#5), isto significa que 45% dos trabalhos classificados aplicam conceitos relacionados às duas abordagens. Os trabalhos #1 e #3 não explicitam o termo orientado a dados, mas utilizam aspectos da abordagem Dc, como especificações em estilo declarativo, estrutura de dados (tabelas, coleções) para expressar metamodelos e modelos. Além disso, o trabalho #1 utiliza uma linguagem declarativa para expressar regras de transformações no âmbito M2M. Já os trabalhos #2 e #3 incorporam a transformação de modelos no processo de desenvolvimento de software, sendo que a transformação M2T é utilizada no processo de desenvolvimento de software para rede de sensores orientada a dados e a transformação de modelos arquiteturais em modelos executáveis. No Trabalho #5, a abordagem orientada a dados é usada como base para um estudo de mapeamentos de esquemas em operações de banco de dados. Os problemas inerentes ao mapeamento são discutidos com propostas de soluções, os quais são semelhantes aos encontrados na MDE.

Tabela 3.3: Classificação de Trabalhos Relacionados à DC

#	Trabalho	Domínio de Problema	Tecnologias	MDE
1	(Batory e Azanza, 2017)	Uma abordagem direcionada à aprendizagem da MDE para estudantes de graduação em computação.	Banco de Dados Relacional, Prolog e Java	√
2	(Thang et al., 2011)	Desenvolvimento de um Framework orientado a modelos (M2T) para o desenvolvimento de aplicações de redes de sensores orientado a dados.	ATL, Eclipse Acceleo, nesC, GMF e TinyOS	√
3	(Almendros-Jiménez et al., 2016)	Uma abordagem para validação e transformação de modelos baseada em programação lógica.	Linguagens Prolog, PTL, ATL e OCL. RDF (*Resource Description Framework*)	√
4	(Ge et al., 2012)	Uma abordagem para transformação de modelos arquiteturais em modelos executáveis, ambos em XML.	Tecnologias XML como: XSD, DTD, XSL e XPath	√
5	(Bonifati et al., 2011)	Um estudo sobre o mapeamento de esquemas em banco de dados relacional.	Datalog, XML, XQuery, XSLT, OWL e sistemas P2P.	√
6	(Shao e Li, 2018)	Uma análise sobre o processamento paralelo de grafos, sob os aspectos pragmático e escalável.	Um conjunto de abordagens e algoritmos, como MapReduce, Pregel, GraphQL, entre outros.	–
7	(Laptev et al., 2016)	Uma abordagem para *Data Stream Management Systems* incluindo uma linguagem declarativa (ESL) e o sistema Stream Mill em modo cliente-servidor.	*client-server*, SQL, C++, ESL	–
8	(Aderholdt et al., 2018)	Implementações de aspectos orientado a dados à SharP, uma biblioteca para gerenciamento de estrutura de dados em *clusters*.	*clusters de nós*, MPI, C++, SharP	–
9	(Liu e Mellor-Crummey, 2013)	Inserção de uma abordagem orientada a dados ao conjunto de ferramentas para análise de desempenho de programas paralelos (HTCToolkit).	*clusters de nós*, Fortran, C, C++, MPI, OpenMP	–
10	(Dolby et al., 2012)	Uma abordagem orientada a dados para sincronização de aplicações paralela em Java.	Processadores multinúcleos, linguagem Java e Anotações Java (AJ)	–
11	(Kim et al., 2019)	Uma abordagem orientada a dados para a computação aproximada por meio da *Data Subsetting* visando desempenho para a computação paralela.	processador multinúcleos, C++, OpenMP	–

Conforme a classificação de trabalhos (#6 a #11) descrita nas linhas 6 a 11 da Tabela 3.3, a abordagem orientada a dados é utilizada nos respectivos domínios: processamento paralelo e escalável de grafos; gerenciamento de sistemas de *streams* em modo cliente-servidor; gerenciamento de estruturas de dados em *clusters*; análise de desempenho de programas paralelos, sincronização de programas paralelos em Java e *Data Subsetting* para desempenho da computação paralela. Esses trabalhos sinalizam que a computação paralela é o ponto relevante dessa seleção e classificação de trabalhos relacionadas à Dc. Além disso, o estilo declarativo está presente nas implementações e discussões das abordagens propostas nesses trabalhos. Porém, pode haver outros trabalhos com domínios de problemas em que a Dc é utilizada e que não foram selecionados para esta pesquisa.

O resultado da classificação de trabalhos apresentado na Tabela 3.3, mostra uma aproximação entre as abordagens Dc e MDE, em que alguns aspectos dessas abordagens são utilizadas em conjunto, sejam aspectos da Dc aplicados em soluções MDE ou o contrário, aspectos da MDE utilizados em aplicações Dc (45% dos trabalhos). Isto pode ser visto como uma estratégia para lidar com diferentes domínios de aplicações que utilizam dados e modelos. As características dessas abordagens podem propiciar a integração entre modelos (heterogêneos) e dados (dados oriundos de diferentes fontes e formatos) em ciclos de desenvolvimento de aplicações baseadas em Big Data, Inteligência Artificial, Iot, entre outras. Combemale et al. (2020) compartilham dessa visão, em que o framework MODA

(*Models and Data*) é proposto para prover a integração das abordagens MDE e Dc voltada ao ciclo de vida de sistemas sócio-técnicos. São considerados sistemas sócio-técnicos aqueles que se preocupam com os aspectos humanos, sociais e organizacionais, tais como os sistemas de transporte inteligente, gerenciamento de desastres, gerenciamento inteligente de energia, gestão de emergências, entre outros. Portanto, a junção de abordagens como a MDE e Dc pode ser considerada uma alternativa viável ao processamento de VLMs e dos respectivos dados.

3.3 CONSIDERAÇÕES FINAIS

Neste capítulo foram apresentados 28 trabalhos relacionados a esta tese, os quais foram selecionados e classificados com base em uma revisão bibliográfica estruturada. A classificação desses trabalhos foi dividida em dois grupos. A composição do primeiro grupo é pautada na busca pelo estado da arte envolvendo a TP/DM, incluindo o particionamento/fragmentação de modelos. O segundo grupo é formado por um conjunto de trabalhos que implementam aspectos da Dc, sobretudo em consonância com a MDE.

O resultado da seleção e classificação de trabalhos do primeiro grupo mostra que o paralelismo implícito combinado com a linguagem ATL por meio de frameworks têm sido até então o mais utilizado para a TP/DMs. No entanto, mecanismos para o particionamento de modelos a fim de melhorar o desempenho do processamento paralelo/distribuído de modelos são pouco explorados nas transformações de modelos, mesmo existindo abordagens dedicadas ao particionamento de grandes modelos para diferentes propósitos. A fragmentação de modelos de entrada não é utilizada como uma estratégia para minimizar o carregamento por completo desses modelos na memória principal do ambiente de execução. Consequentemente, possibilitar o processamento paralelo/distribuído dos fragmentos resultantes da fragmentação. As características do modelo de entrada e o domínio de transformação, que diz respeito à complexidade estrutural, não são avaliadas pelos trabalhos que executam a transformação de modelos. O resultado da classificação do segundo grupo de trabalhos mostra que aspectos das abordagens Dc e MDE são combinados e aplicados em diferentes domínios. O processamento paralelo é uma estratégia comum entre os trabalhos desse grupo, como também o uso acentuado do paralelismo implícito em conjunto com implementações declarativas para o processamento de dados, independente do domínio da aplicação.

Os principais aspectos da abordagem Dc4MT como: Fragmentação de Modelos de Entrada (FME), Particionamentos Implícito (PIM) e Explicito (PEM) de Modelos, Transformação Paralela e Distribuída de Modelos (TPD), Extração de Modelos para Grafo (EMG), o uso de modelos nos Formatos JSON e XMI (FJX), a visualização da Complexidade Estrutural de Modelos de entrada (CEM) e as Abordagens Declarativa e Relacional (ADR) são comparados com um subconjunto dos trabalhos classificados nas Tabelas 3.1 e 3.2. Essa comparação é apresentada na Tabela 3.4, onde os acrônimos dos aspectos da Dc4MT caracterizam as colunas dessa tabela. A presença desses aspectos no trabalho é assinalada com "$\surd$", enquanto que, a ausência é assinalada com "-".

Tabela 3.4: Comparação de Aspectos da Dc4MT com um Subconjunto dos Trabalhos Relacionados

Trabalhos	FME	PIM	PEM	TPD	EMG	FJX	CEM	ADR
(Benelallam, 2016)	-	√	√	√	-	-	-	√
(Aracil e Ruiz, 2017)	-	√	√	√	-	-	-	√
(Benelallam et al., 2015)	-	√	-	√	-	-	-	√
(Burgueño, 2016)	-	√	-	TP	-	-	-	-
(Fekete e Mezei, 2016)	-	√	-	TP	√	-	-	-
(Benelallam et al., 2018)	-	√	-	√	-	-	-	√
(Tisi et al., 2013)	-	√	-	TP	-	-	-	√
(Imre e Mezei, 2012)	-	-	-	TP	-	-	-	-
(Daniel et al., 2017)	-	-	-	√	-	-	-	-

***TP** Transformação Paralela

De acordo com o resultado das classificações de trabalhos apresentados neste capítulo, e da comparação contida na Tabela 3.4, percebe-se que ainda há lacunas a serem preenchidas no que diz respeito ao desempenho e escalabilidade à TM. Isto posto, a abordagem Dc4MT busca preencher uma parte dessas lacunas com a fragmentação, extração e o processamento paralelo e distribuído de TMs. No próximo capítulo, a abordagem Dc4MT é descrita, incluindo a visualização de complexidade estrutural dos modelos de entrada em uma plataforma escalável de paralelismo implícito.

4 A ABORDAGEM DC4MT

A Dc4MT (Data-centric for Model Transformation) é uma abordagem escalável para transformação paralela/distribuída de modelos. A abordagem provê um conjunto de operações para Fragmentação, Extração e Transformação de VLMs por meio de uma plataforma de distribuição implícita de dados. No âmbito da MDE, a transformação de VLMs e a escalabilidade são questões que ainda não estão totalmente resolvidas, visto que as abordagens tradicionais não dispõem de mecanismos adequados para tratar de tais questões (Tisi et al., 2013; Burgueño, 2016; Benelallam et al., 2018; Daniel, 2017). O conjunto de operações e os aspectos da abordagem Dc4MT são apresentados nas seções deste capítulo.

4.1 VISÃO GERAL DA ABORDAGEM DC4MT

A Dc4MT está estruturada em módulos de modo que se possa implementá-la em diferentes plataformas como o MapReduce (Dean e Ghemawat, 2008), Hadoop (Apache, 2019c), Bloom (BOOM, 2013), entre outras. Nesta tese, a arquitetura modular da abordagem Dc4MT é implementada sob o framework Spark em um nó Master e em nós Workers. Uma ilustração dessa arquitetura é mostrada na Figura 4.1. No nó Master é executada a fragmentação de modelos de entrada por meio do módulo **Fragmentador**. O módulo Fragmentador é acionado pelo módulo Plano de Execução, o qual segue o procedimento definido pela configuração do ambiente. As operações especificadas para os módulos e a configuração do ambiente são utilizadas pelo ambiente de execução, neste caso o framework Spark, para o processamento da transformação de modelos.

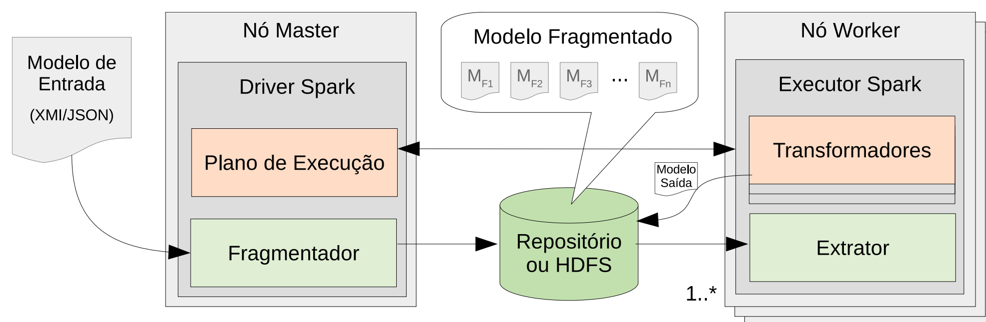


Figura 4.1: Uma Visão Geral de uma Implementação Modular da Dc4MT

O módulo Fragmentador é o responsável por executar a fragmentação do modelo de entrada, gerando fragmentos desse modelo (M_{F1}, M_{F2}, ...), os quais são instanciados em um repositório (execução local) ou em um HDFS (execução distribuída). A fragmentação e a interface com o ambiente de execução são executadas pelo Driver Spark no Nó Master. O módulo **Extrator** tem o papel de processar a extração dos fragmentos do modelo de entrada, traduzindo esses fragmentos para um grafo. Já o módulo **Transformadores** é o responsável por transformar modelos de entrada representados em grafo para modelos de saída. Ambos, são executados em paralelo pelos Executores nos Nó(s) Worker(s).

Na Figura 4.2, é apresentado um conjunto de passos com as suas principais atividades, esses passos abstraem o fluxo do processo de transformação paralela/distribuída de modelos provido pelas operações da abordagem Dc4MT. No Passo 1, a transformação de

modelos é submetida ao ambiente de execução pelo(a) desenvolvedor(a) com a parametrização do ambiente de execução (detalhes da parametrização são apresentados no próximo capítulo). Já no Passo 2, a submissão da transformação é recebida no Nó Master, onde a fragmentação do modelo de entrada é executada. Em seguida, os fragmentos resultantes da operação Fragmentação são distribuídos aos Nós Workers (Nw-1,..., Nw-n) pelo Nó Master no Passo 3. No Passo 4, a extração é executada no(s) Nó(s) Worker(s), traduzindo os fragmentos do modelo de entrada em um grafo direcionado ($G(V, E)$). Inclui na operação de extração a resolução de referências entre os elementos do modelo. Por fim, no Passo 5 a transformação de modelos é executada sobre o grafo, transformando os elementos do modelo de entrada estruturados em vértices (V) e arestas (E) em modelo de saída, o qual é instanciado pelo Nó Master.

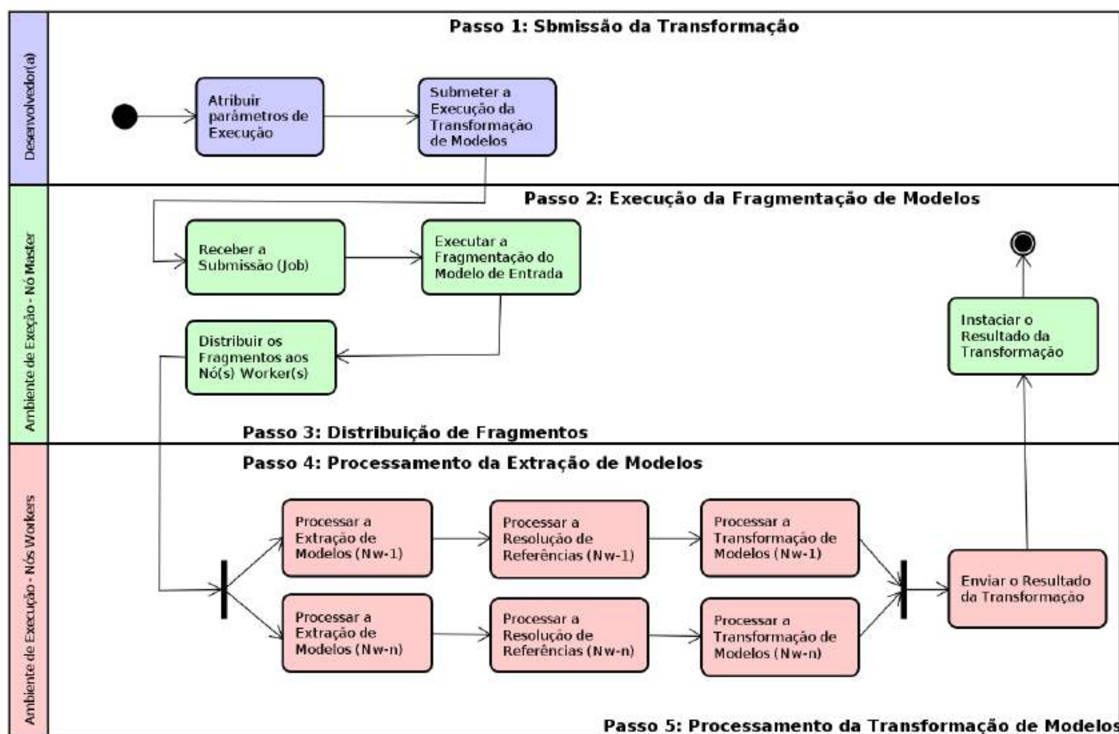


Figura 4.2: Passos das Principais Atividades das Operações Dc4MT

Uma visão geral da Dc4MT é mostrada nas Figuras 4.1 e 4.2, as quais respectivamente representam as perspectivas arquitetural e comportamental do fluxo de operações sobre modelos da abordagem Dc4MT. Para sustentar o fluxo de operações da Dc4MT, um conjunto de algoritmos foi elaborado para garantir que cada operação fornecerá os elementos necessários para o passo seguinte durante a execução desse fluxo. A operação Extração utiliza três algoritmos dedicados à tradução dos fragmentos do modelo de entrada para um grafo, incluindo a resolução de referências. A operação Transformação é especificada em modo declarativo utilizando filtros e regras de transformação.

Na próxima seção, o modelo de comunicação do ambiente de execução é descrito sob a perspectiva de dependência de dados. Esse modelo é essencial para as operações de Extração e Transformação de modelos executadas nos Passos 4 e 5 (Figura 4.2). Nas seções subsequentes, as operações Dc4MT são descritas sob uma perspectiva de implementação.

4.2 MODELO DE COMUNICAÇÃO DO AMBIENTE DE EXECUÇÃO

Todo dado de entrada processado em um ou mais nós Workers são particionados pelo framework Spark. Isto é necessário para garantir a distribuição e a resiliência (RDD) durante o processamento paralelo de dados pelos nós Workers. Um RDD possui as seguintes propriedades: uma lista de partições de objetos para formar o RDD, uma função para iterar sobre essas partições (dados) e uma lista de dependência pai-filho (*parent-child)* entre os RDDs. Um DataFrame é uma abstração de um RDD e uma partição de DataFrame ou GraphFrames é formada por uma coleção de linhas que cabe em um nó de uma máquina ou de um *cluster*, representando a distribuição física de dados entre os nós do ambiente durante a execução (Chambers e Zaharia, 2018; Karau e Warren, 2017).

O tamanho de uma partição é limitada à disponibilidade de memória de um executor detectada pelo particionador do Spark. O particionador também considera a origem do dado de entrada para definir o tamanho de uma partição. Por exemplo, se o dado de entrada está em um sistema de arquivo distribuído como o HDFS, o tamanho da partição será de 64 ou 128 MB (conforme a versão do HDFS). O Spark permite a customização explicita do particionador por meio de métodos como `Object.hashCode` para um particionamento baseado em chaves *hash* (*Hash Partitioning*) ou utilizando o `repartitionByRange` para um particionamento baseado em ordenação de chaves (*Range Partitioning*) quando se sabe como as chaves estão distribuídas ou sequenciadas. Além disso, há fatores que podem influenciar na escolha de um particionamento explicito, tais como: a disponibilidade de recursos, fonte externa onde está o dado de entrada (sistema de arquivos), comandos que geram RDDs (e.g., `join, sort, reduceByKey, ...`), entre outros fatores. Na implementação da Dc4MT não é adotado um particionador explicito de dados, mas é usado uma parametrização que interfere na operação `shuffle` em decorrência do uso de comandos que geram RDDs.

O estilo declarativo é um dos aspectos da Dc4MT em que comandos como `filter`, `join` e `union` são utilizados em conjunto com funções na implementação dos módulos que compõem a Dc4MT (Figura 4.1). Esses comandos interferem na troca de mensagens entre os nós do ambiente quando são executados. Na Figura 4.3, um fluxo de comunicação é ilustrado como um exemplo. Nesse exemplo, as partições do modelo de entrada `Families`, representadas em um GraphFrames, contendo vértices e arestas (`gf.vertices` e `gf.edges`) são submetidas ao ambiente de execução.

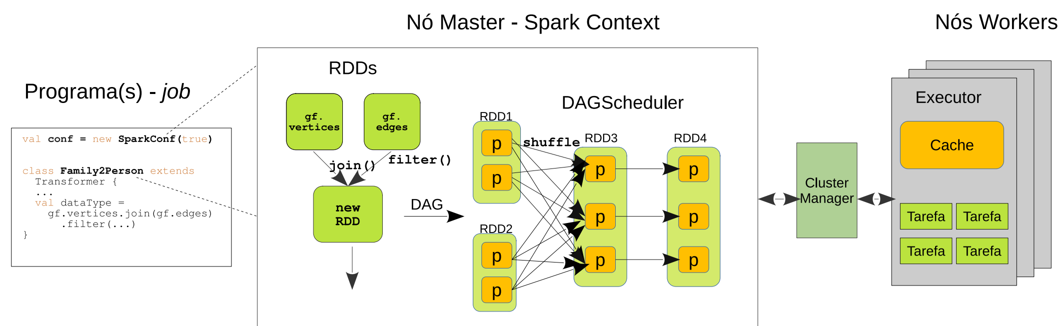


Figura 4.3: Um Exemplo do Fluxo do Modelo de Comunicação

De acordo com a Figura 4.3, o fluxo de comunicação entre as partições de dados é formado quando um ou mais programas são submetidos (*job*) ao ambiente de execução, neste caso ao Nó Master, caracterizando o Spark Context. Os comandos contidos no programa submetido para execução formam um grafo direcionado acíclico (DAG), em que os

comandos representam as arestas e os RDDs os vértices. No exemplo da Figura 4.3, o DAG é formado por um trecho do programa que específica a transformação `Family2Person`. O GraphFrames contendo os DataFrames `gf.vertices` e `gf.edges` formam os vértices representando os RDDs e os comandos `filter()`) e `join()` formam as arestas. O resultado do comando `join()` gera um novo RDD (new RDD). O DAG representa a execução lógica de comandos utilizados nas operações de Transformações e Ações no Spark, visto que os vértices representam os RDDs e as arestas representam os comandos de uma Transformação ou Ação. Enquanto comandos de Ação, como `write`, `show` não são invocados, o fluxo de comunicação se restringe em gerar o DAG em decorrência do aspecto *Lazy-evaluate* presente no framework Spark.

A presença de um comando de Ação no programa faz com que o DAG seja convertido em um plano físico de execução otimizado pelo Spark. O DAG em um plano físico de execução é dividido em estágios de tarefas, os quais são submetidos pelo DAGScheduler ao Cluster Manager. O DAGScheduler é uma camada de *scheduling* do framework Spark responsável por garantir as execuções orientadas a estágios, em que um plano de execução lógico é transformado em um plano de execução físico em estágios (Apache, 2019d). As tarefas são escalonadas e lançadas aos Executores (Nós Workers) via Cluster Manager para processar as partições `p` de dados contidas nos RDDs. Quando os Executores concluem a execução de todas as tarefas o resultado é enviado de volta via o Cluster Manager ao nó Master. Pode-se observar no DAGScheduler a representação do envio de mensagens entre as partições dos RDDs em relação aos comandos `join` e `filter`. O uso de *joins* pode disparar a operação `shuffle`, em que a troca de mensagens entre as partições é de um-para-muitos. Já o comando `filter` exige a troca de mensagens entre as partições de um-para-um. Nesse exemplo, utilizou-se de um programa que manipula um GraphFrames para demonstrar o fluxo do modelo de comunicação no Spark. No entanto, qualquer operação executada nos nós Workers é dependente de RDDs no que diz respeito a troca de mensagens entre os executores (nós Workers) envolvidos no paralelismo implícito.

As operações de Transformação em Spark podem ter dois tipos de dependência entre as partições *parent-child*: *Narrow* ou *Wide*. Por exemplo, a execução de junções (`join`) requer que as chaves correspondentes a cada RDD estejam localizadas na mesma partição para que essas chaves possam ser combinadas no mesmo nó, caracterizando uma transformação *Narrow*. Do contrário, a operação `shuffle` é realizada pelo Spark de modo que os RDDs compartilhem um particionador (e.g., chave) e os dados com as mesmas chaves são alocados nas mesmas partições, caracterizando uma transformação *Wide*. Se os RDDs têm o mesmo particionador, suas partições podem ser co-alocados para minimizar a movimentação de dados entre os RDDs. Na Figura 4.4, um exemplo dos tipos de dependência entre partições *parent-child* contidas nas RDDs 1, 2 e 3 é mostrado. No tipo *Narrow*, cada partição das RDDs *parent* (1 e 2) usa no máximo uma partição do RDD *child*, permitindo em um nó a execução em *pipeline*[1]. A dependência é *Wide* quando múltiplas partições *child* depende de uma ou mais partições *parent*, requisitando que os dados das partições *parent* estejam disponíveis para a operação `shuffle` entre os nós do ambiente de execução.

[1] *pipeline* é um conjunto de elementos conectados em série, em que a saída de um elemento é a entrada do próximo elemento em uma operação de processamento de dados.

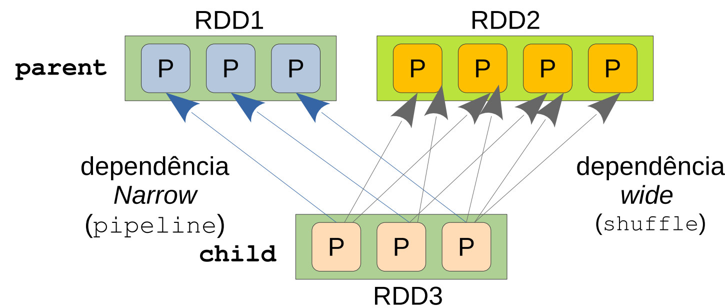


Figura 4.4: Tipos de Dependência entre Partições RDDs

Mesmo com a otimização do plano de execução do Spark, o processamento de junções tende a ter dependências *Wide*. A ocorrência de dependência de dados do tipo *Wide* diminui o desempenho do processamento em relação a dependência do tipo *Narrow*. A estratégia de co-alocação de partições pode alterar uma dependência do tipo *Wide* para o tipo *Narrow*. Para tanto, ao menos um dos RDDs envolvidos em uma junção de dados (*join*) têm que caber na memória de um nó Worker. Quando isto é possível, o RDD que cabe em memória é replicado entre os nós Workers e o processamento do `join` será local e não incluirá o custo da operação `shuffle`, apenas o custo de replicação. Por exemplo `bigDF.join(broadcast(smallDF)`, nesse exemplo o DataFrame `smallDF` será co-alocado próximo às partições do DataFrame `bigDF` por meio da função `broadcast` do framework Spark. A estratégia de replicação (`broadcast`) de RDDs não foi adotada na implementação da Dc4MT, uma vez que os DataFrames `gf.vertices` e `gf.edges` que compõem o GraphFrames possuem tamanhos semelhantes. Além disso, se um RDD replicado para os nós (máquina/cluster) não couber na memória de um desses nós, o Spark ignora a função `broadcast` e executa a operação `shuffle`, juntando as partições de dados contidas no GraphFrames (RDDs). Assim, os dados são disponibilizados aos Executores (Workers) alimentando a execução paralela de tarefas em uma dependência de dados tipo *Wide*.

Além da fragmentação, não há na Dc4MT uma estratégia de particionamento explícito de modelos de entrada durante a execução das operações de extração e transformação. Ao invés disso, é adotada uma parametrização do particionamento de dados durante a operação `shuffle` realizada pelo Spark, provocando alterações nesse particionamento buscando a melhoria de desempenho. O valor padrão do parâmetro `shuffle.partitions` é de 200 partições de saída para cada operação `shuffle` (Apache, 2019d). Segundo Chambers e Zaharia (2018), o parâmetro `shuffle.partitions` pode ser configurado de modo que *datasets* pequenos ($<$ 256 MB) não sejam particionados em 200 partições (particionamento padrão) durante as operações `shuffle`, impactando no desempenho do processamento desses *datasets*. Assim, recomenda-se que para data sets pequenos o valor do parâmetro seja a quantidade de nós físicos + 1 ($N_{no} + 1$). Já para data sets médios e grandes ($>$ 1 GB) o valor do parâmetro deve ser maior que 200. Tendo em vista as recomendações de Chambers e Zaharia (2018), as parametrizações utilizadas nos experimentos para validar a Dc4MT em modo local e distribuído são apresentadas no próximo capítulo. Na próxima seção, a operação Fragmentação é descrita.

4.3 FRAGMENTAÇÃO DE MODELOS

A fragmentação de modelos adotada nesta tese leva em conta o aspecto da disjunção entre os fragmentos. Isso significa que os fragmentos são formados por elementos únicos gerados pela operação de fragmentação. A representação da fragmentação e desse aspecto podem ser definidas da seguinte maneira:

$$\forall E_M \in \{M_e\} : \{E_M\} \mapsto \{\{F_{E_1}\}, \{F_{E_2}\}, \{F_{E_n}\}\} \Rightarrow \{F_{E_1}\} \cap \{F_{E_2}\} \cap \{F_{E_n}\} = \varnothing$$

Qualquer elemento E_M do modelo de entrada M_e é submetido a operação de fragmentação tal que todos elementos $\{E_M\}$ são mapeados para os fragmentos $\{F_{E_n}\}$, onde cada fragmento possui elementos distintos. Além da disjunção, a ideia de *proxy* e a alternativa de fragmentação *top-down* são adaptadas de Amálio et al. (2015). *Proxy* é usado como um ponto de referência entre os elementos do mesmo fragmento e a *top-down* indica que há continuação (*continuations*) do modelo entre os fragmentos (do elemento raiz até o último elemento da hierarquia). Para complementar a estratégia de fragmentação, são observadas as discussões de Scheidgen et al. (2012) sobre a indexação de elementos fragmentados e o tamanho de cada fragmento a ser gerado. Os autores afirmam que não há um tamanho arbitrário para um fragmento, a atribuição do tamanho está associada ao ambiente em que os fragmentos serão processados e ao tipo de modelo a ser fragmentado. De acordo com Gulati e Kumar (2017), o tamanho de blocos de dados gerados pelo HDFS são 32 MB para execução local e 64/128 MB (dependendo do versão do Hadoop) para execução distribuída. As técnicas que compõem a estratégia de fragmentação de modelos de entrada proposta para a Dc4MT foram adaptadas dos trabalhos de Amálio et al. (2015) e Scheidgen et al. (2012) (trabalhos 12 e 13 classificados nas Tabelas 3.1 e 3.2).

No Algoritmo 1, há uma representação da estratégia de fragmentação adotada na Dc4MT, a qual é instanciada no módulo Fragmentador. De acordo com Algoritmo 1, as variáveis Ie_{atual}, $Frag$, T_{Frag} e T_{Fmin} são atribuídas (linhas 1 a 7) ao controle da fragmentação. Os elementos do modelo de entrada são indexados (Ie_{atual}) sequencialmente a partir do elemento raiz do fragmento. O índice do elemento raiz é usado como um *proxy* para permitir o acesso aos demais elementos do fragmento, como também indicar o ponto de conexão entre os fragmentos (e.g., Figura 4.5 arquivos 0,6,8). Nas linhas 5 e 7 o tamanho mínimo do fragmento é atribuído (via ambiente de configuração). Para a execução local o tamanho mínimo é de 32 MB e de 128 MB para a execução distribuída em HDFS. Para todo elemento do modelo em fragmentação é associado a um fragmento por meio da função `escrever` por meio dos argumentos `Elem` e `Frag` (linha 14). O aspecto de disjunção da estratégia de fragmentação sobrepõem o tamanho mínimo do fragmento, em que um elemento do modelo não pode pertencer a dois ou mais fragmentos. Isso significa que enquanto um elemento não for totalmente processado (inclui atributos do elemento) a inserção de elementos no fragmento não é interrompida (linhas 16 a 18), mesmo que o tamanho do fragmento (T_{Frag}) já tenha atingido o tamanho atribuído (T_{Fmin} 32 ou 128 MB).

Na Figura 4.5, é ilustrado o fluxo da operação de fragmentação de modelos, em que o modelo de entrada (*input_model*) contido em um Repositório ou HDFS é fragmentado pelo módulo Fragmentador. Na medida em que cada fragmento é formado, esses fragmentos são organizados em um diretório de arquivos (*input_model.fragmentado*). Nesse diretório, os fragmentos são depositados e recebem como nome o índice *proxy* (e.g., 0, 6, 8).

Algoritmo 1 Fragmentação de Modelos

Require: $M_{entrada} \neq \emptyset$
1: $Ie_{atual} \leftarrow -1$ // índice do elemento atual
2: $Frag \leftarrow null$ // fragmento atual
3: $T_{Frag} \leftarrow 0$ // tamanho do fragmento atual
4: **if** $hdfsUpload \leftarrow false$ **then**
5: $T_{Fmin} \leftarrow 32$ MB // execução local
6: **else**
7: $T_{Fmin} \leftarrow 128$ MB // execução distribuída
8: **end if**
9: **for all** $Elem \in M_{entrada}$ **do**
10: $Ie_{atual} \leftarrow Ie_{atual} + 1$
11: **if** $Frag \leftarrow null$ **then**
12: $Frag \leftarrow$ novo_fragmento(Ie_{atual})
13: **end if**
14: escrever($Elem$, $Frag$)
15: $T_{Frag} \leftarrow T_{Frag} +$ tamanho($Elem$)
16: **if** $T_{Frag} \geq T_{Fmin}$ **then**
17: fechar($Frag$)
18: $Frag \leftarrow null$
19: **end if**
20: **end for**

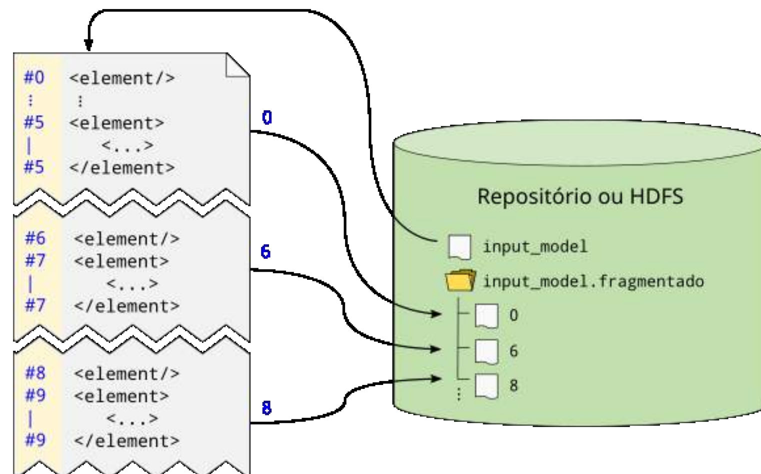


Figura 4.5: Fluxo da Operação de Fragmentação de Modelos

Nas Listagens 4.1 e 4.2, há um exemplo do resultado da fragmentação de modelos. Um trecho do modelo de entrada `Families` representa o modelo a ser fragmentado (*input_model*). O resultado da fragmentação é representado por um trecho de fragmento mostrado na Listagem 4.2. Nessa listagem, as *tags* $< inject_Root >$ e $<\backslash inject_Root >$ são exibidas. No entanto, essas *tags* são descartadas durante a operação de Extração, uma vez que elas não fazem parte do modelo de entrada.

Listagem 4.1: Modelo de Entrada `Families`

```
<?xml version="1.0"
    encoding="ISO-8859-1"?>
<xmi:XMI xmlns="Families">
<Family lastName="March">
  <father firstName="Jim"/>
  <mother firstName="Cindy"/>
  <sons firstName="Brandon"/>
 </Family>
  <Family lastName="Sailor">
    <father firstName="Peter"/>
       ...
 </Family>
  <Family lastName="Camargo"/>
      ...
 <\xmi:XMI>
```

Listagem 4.2: Exemplo com dois Fragmentos

```
<inject_root>
<Family lastName="March">
  <father firstName="Jim"/>
  <mother firstName="Cindy"/>
  <sons firstName="Brandon"/>
 </Family>
  <Family lastName="Sailor">
    <father firstName="Peter"/>
    <mother firstName="Jackie"/>
          ...
 </inject_root>
 <inject_root>
    <Family lastName="Camargo"/>
    ...
</inject_root>
```

A estratégia de fragmentação adotada e descrita nesta seção permite uma fragmentação balanceada de modelos, por meio de atribuição do tamanho para cada fragmento. A indexação de elementos dos modelos e os *proxies* habilitam a execução paralela/distribuída dos fragmentos de maneira consistente. Na próxima seção, a operação Extração de modelos é descrita.

4.4 EXTRAÇÃO DE MODELOS

A extração de modelos é uma operação que recebe os fragmentos do modelo de entrada e os traduz para um grafo por meio da execução do módulo Extrator. A tradução é necessária para que o modelo de entrada possa pertencer ao domínio de modelagem requerido pela Dc4MT, uma vez que os modelos de entrada pertencem a espaços técnicos diferentes. Além disso, quando o modelo de entrada é traduzido para o formato grafo as possibilidades de computação sobre esse modelo podem ser ampliadas. Na Dc4MT, um grafo é representado por meio de um GraphFrames. Nesta seção, são descritas a extração de modelos, resolução de referências entre os elementos do modelo de entrada e a visualização da densidade de referências entre elementos do modelo.

O módulo Extrator é executado no nó(s) Worker(s) para processar em modo paralelo os fragmentos do modelo de entrada, os quais são distribuídos pelo nó Master, essa distribuição pode ser representada da seguinte forma:

$$\forall N_W \subset A_X : A_{X_{(N_{W_0}, N_{W_1}, N_{W_n})}} \leftarrow N_{M\{F_{E_1}, F_{E_2}, F_{E_n}\}}$$

em que cada nó Worker (N_W) contido no ambiente de execução (A_X) pode ser utilizado para formar o ambiente paralelo/distribuído de execução ($A_{X_{\{N_{W_0}, N_{W_1}, \ldots\}}}$). Nesse ambiente, a operação Extração é executada quando o nó Master atribui os fragmentos ($N_{M\{F_{E_1}, ..\}}$) aos nós Workers (N_W). Na Figura 4.6, há um modelo que representa a operação de extração. Nessa representação um modelo de entrada em XMI é traduzido para um grafo. No resultado da representação há um elemento raiz (`root`), do qual são derivados dois elementos filhos (`child`) ligados respectivamente pelas arestas 0 e 1. O elemento `child` ligado pela aresta 0 possui o atributo `name="A"` e uma referência (`ref="/1"`) ao segundo elemento `child` que contem o atributo `name="B"` e os elementos filhos `detailOne` e `detailTwo`. Durante a extração, cada elemento do modelo de entrada é traduzido para um vértice no grafo contendo um `id` e o `nome` do elemento como conteúdo. A aresta que conecta um elemento a seu pai contém sua chave (e.g., 0,1). Os atributos dos elementos também são convertidos em vértices do grafo cujo conteúdo é o valor do atributo (`A`, `B`, ...). A aresta que conecta o vértice de um atributo ao elemento que o contém possui como chave

o nome do próprio atributo (neste caso `name`). Os elementos que possuem referências com o `"/1"` para indicar a posição do elemento referenciado em uma hierarquia *containment* são tratados como índices e denotam o n-ésimo filho do elemento referenciado.

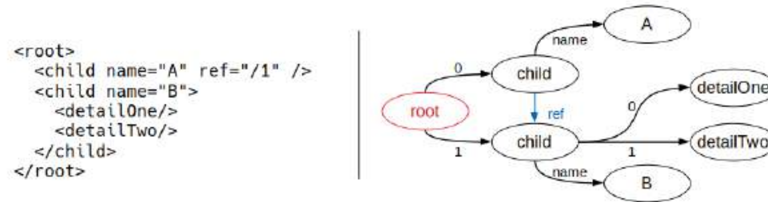


Figura 4.6: Modelo para Tradução de Modelos em Grafos

O processamento paralelo/distribuído dos fragmentos é realizado por meio da API RDD, em que o modelo fragmentado é tratado como uma lista de tuplas $\{Nome, Bytes\}$, cada uma correspondendo a um fragmento. O elemento $Nome$ contêm o caminho do arquivo correspondente ao fragmento e o índice do primeiro elemento do fragmento. O elemento *Bytes* é o conteúdo do fragmento tratado como um fluxo de bytes. Para cada fragmento é executada a operação Spark `flatMap()`. Essa operação consiste em mapear os elementos dos fragmentos para novos RDDs (RDD_{elems}), os quais são usados para gerar o GraphFrames. Os Algoritmos 2 e 3, constituem uma representação do módulo Extração (Model2GraphFrames). No Algoritmo 2 é representado o mapeamento dos elementos do modelo de entrada (fragmentos) aos RDDs. Nesse mapeamento, a estrutura hierárquica dos elementos é preservada independente da ordem em que os fragmentos chegam aos executores (nós Workers).

Os fragmentos do modelo são lidos em *streaming* por um parser em modo paralelo (linhas 2-7 do Algoritmo 2). O parser adotado para a leitura depende do formato do modelo de entrada. Uma vez que o $Parser$ é inicializado com o fluxo de bytes, em que o fragmento é lido de maneira sequencial, gerando eventos para o início ($InicioTag$) e fim ($FinTag$) de um elemento e seu(s) atributo(s). Um fluxo de eventos (*Evt*) representa a estrutura do fragmento enviado pelo parser (`stream.reader`). Ao processar esses eventos é possível obter do parser informações de cada elemento em tempo de execução e reconstruir a estrutura desses elementos, inserindo índices aos elementos e aos seus atributos. A função `criarElemento()` (linhas 31-34) é responsável pela reconstrução dos elementos em RDDs. Por meio dos IDs é possível distinguir elementos pais dos elementos filhos. Por exemplo, na linha 11 a pilha de IDs está vazia, isso significa que o elemento raiz do modelo foi lido e ele será criado pela função `criarElemento`. Nesse caso, a quadrupla da linha 12 (Id_{atual}, *NomeElemento*, -1, Id_{atual}) é usada como argumento da função. Nessa quadrupla o valor `-1` é usado para distinguir do índice inicial (Id_{atual}) com valor 0. Em seguida, os atributos do elemento raiz são gerados com a função `gerarId()` e os argumentos, *valorAtrib*, *IDs.topo*, *nomeAtrib* (linha 14) de modo que os índices dos atributos não apresentem duplicidades. O Id do elemento é passado aos atributos pelo argumento *IDs.topo*, enquanto que o valor e o nome dos atributos são constituídos respectivamente pelos argumentos *valorAtrib* e *nomeAtrib*. O mesmo procedimento é adotado para gerar elementos filhos e os atributos correspondentes (linhas 19-23).

Algoritmo 2 Mapeamento dos Fragmentos do Modelo Entrada Para RDDs

```
Require: RDD_modelo ≠ ∅
 1: for all Frag ∈ RDD_modelo do
 2:    Nome ← Frag.Nome
 3:    Bytes ← Frag.Bytes
 4:    Id_inicial ←obterId(Nome)
 5:    Id_atual ← Id_inicial
 6:    Parser ← parser(Bytes)
 7:    IDs ← stak(long)
 8:    for all Evt ∈ Parser do
 9:       if Evt = InicioTag then
10:          NomeElemento ← Parser.nomeAtual
11:          if IDs = ∅ then
12:             criarElemento(Id_atual, NomeElemento, -1, Id_atual)
13:             for all Atributo ∈ Parser.atributos do
14:                criarElemento(gerarId(), valorAtrib, IDs.topo, nomeAtrib)
15:             end for
16:             IDs := IDs + Id_atual
17:             Id_atual ← Id_atual + 1
18:          else
19:             Id_Filho ← gerarId()
20:             criarElemento(Id_Filho, NomeElemento, IDs.topo, Id_atual)
21:             for all atributo ∈ Parser.atributos do
22:                criarElemento( gerarId(), valorAtrib, Id_Filho, nomeAtrib)
23:             end for
24:          end if
25:       else if Evt = fimTag then
26:          if IDs ≠ ∅ then
27:             IDs := IDs − IDs.topo
28:          end if
29:       end if
30:    end for
31:    function criarElemento()
32:    RDD_elems ← flatMap()
33:      return RDD_elems
34:    end function
35: end for
```

A representação da atribuição dos RDD_{elems} ao GraphFrames é exibida pelo Algoritmo 3 como parte final da representação da extração paralela/distribuída de modelos. Uma vez que os elementos do modelo de entrada estão espalhados em RDDs ($RDDelems$) é necessário organizá-los em DataFrames para formar o GraphFrames (GF). Para isto, os RDDs são processadas de modo que as listas de atributos que formam as arestas e vértices sejam separadas e assinaladas aos respectivos DataFrames. O DataFrame *arestasDF* recebe listas de atributos para formar as arestas do GF. Esses atributos são passados à função `toDF` do framework Spark como argumentos para as colunas `"src", "dst"` e `"key"`, as quais são atribuídas ao DataFrame *arestasDF*. Para o processamento paralelo que origina o DataFrame *verticesDF* é necessário obter o elemento raiz (`root`) do contexto do Spark (`SparkContext`) e atribuir a um RDD (*raizRDD*) (linha 2). Esse RDD é criado por meio do método `parallelize` que transmite os valores `-1, "root", null` e `null` já que o RDD_{elems} é formado pela quadrupla `"id", "value", "parent"` e `"key"`, de modo que cada elemento do modelo contenha respectivamente um índice, valor, índice do seu elemento pai e uma chave. Um DataFrame com vértices temporários ($verticesTempDF$) é criado com o resultado da união dos RDDs *raizRDD*

e RDD_{elems}. Da quadrupla, somente as colunas `"id"` e `"value"` são utilizadas para compor o DataFrame *verticesDF* (`drop("parent", "key")` (linha 4 do Algoritmo 3). Por fim, os DataFrames *verticesDF* e *arestasDF* são assinalados ao *GF* por meio da classe `GraphFrame` da API GraphFrames.

Algoritmo 3 Criação do GraphFrames a partir de RDD_{elems}

Require: $RDD_{elems} \neq \emptyset$
1: $arestasDF \leftarrow List(RDD_{elems})$.`toDF("dst", "src", "key")`
2: $raizRDD \leftarrow$ `parallelize(-1, "root", null, null)`
3: $verticesTempDF \leftarrow (raizRDD \cup RDD_{elems})$
4: $verticesDF \leftarrow verticesTempDF$`.drop("parent", "key")`
5: $GF \leftarrow$ `GraphFrame(` $arestasDF$, $verticesDF$ `)`

As referências hierárquicas entre os elementos do modelo de entrada são preservadas pelos Algoritmos 2 e 3 na criação do GraphFrames (*GF*). No entanto, as referências entre os elementos além da hierarquia *containment* não são resolvidas, apenas são gerados vértices e arestas para esses elementos com o conteúdo das respectivas referências. Por exemplo, um vértice com o `id = 1048844` e `value = "/2"` e uma aresta com `src = 1048841`, `dst = 1048844` e `key = "type"` significa que somente a referência `"/2"`foi mapeada para o tipo (`type`), mas ainda não há como estabelecer uma ligação (aresta) para o valor dessa referência. A resolução de referências entre elementos do modelo é apresentada a seguir.

4.4.1 Resolução de Referências

Além da hierarquia *containment* de elementos (do elemento raiz ao(s) elemento(s) folha), há modelos que contêm elementos com uma ou mais referências a outro(s) elemento(s) independente de sua posição hierárquica no modelo. Por exemplo, o tipo (`type`) de um elemento pode possuir uma referência `/1` (`type="/1"`). Essa referência pode ser direcionada a um elemento do tipo `DataType`, cuja hierarquia para os elementos desse tipo está organizada da seguinte forma: posição 0 para `Decimal`, posição 1 para `Character` e posição 2 para `Integer`. Neste caso, a Resolução de Referências busca substituir a referência `/1` por `Character`, criando novos vértices e arestas para o GraphFrames. As referências em que os vértices possuem valores (`value`) com estruturas em *arrays* também são tratados, derivando novos vértices e arestas. Modelos que utilizam UUID (*Universally Unique Identifier*)[2] ainda não são suportados pelo módulo Extrator.

O Algoritmo 4, representa a transformação de arestas e vértices que possuem referências em novos subgrafos, integrando esses subgrafos ao GraphFrames. As arestas que identificam os elementos do modelo com referência(s) são selecionadas (linhas 1 e 2 do Algoritmo 4) e utilizadas em uma junção com os vértices do GraphFrames (*GF*) para formar um subgrafo (Sub_{GF}). Desse subgrafo, os elementos são extraídos formando novos vértices e novas arestas (linhas 3 à 6). A extração (`extrairElms`) trata a lista de `referencias` e identifica cada elemento referenciado, preservando os respectivos valores dos `ids` contidos nos vértices do Sub_{GF}. Esses `ids`, são utilizados como base para a criação de novos vértices e arestas (linhas 5 e 6). Os vértices contidos no *GF* e utilizados na formação do $Sub_{GF}.vertices$ são excluídos (linha 8) para não incidir sobre os vértices do GF final e reproduzir duplicação de `ids`. Assim, vértices e arestas contidas no

[2]Um identificador universal de 128-bit usado para distinguir informação em sistemas computacionais.

GF são agrupados com os novos vértices e novas arestas nos DataFrames $verticeFinal$ e $arestasFinal$, formando um novo GF (linha 10).

Algoritmo 4 Representação da Resolução de Referencias

Require: $GF \neq \emptyset$
1: $referencias \leftarrow List(String)$
2: **for all** $arestas \in GF.edges(referencias)$ **do**
3: $Sub_{GF} \leftarrow (GF.edges(referencias) \bowtie GF.vertices)$
4: $elms \leftarrow$ `extrairElms`(Sub_{GF})
5: $novosVerticesDF \leftarrow$ `criarVertices`$(elms)$
6: $novasArestasDF \leftarrow$ `criarArestas`$(elms)$
7: **end for**
8: $verticesFinal \leftarrow ((GF.vertices - Sub_{GF}.vertices) \cup novosVerticesDF)$
9: $arestasFinal \leftarrow (GF.edges \cup novasArestasDF)$
10: $GF \leftarrow$ GraphFrame($verticesFinal$, $arestasFinalDF$)

Após a resolução de referências, a tradução do modelo para o grafo é finalizada e os elementos do modelo são atribuídos ao GraphFrame (GF). Isso significa que a resolução de referências incrementa o GraphFrames resultante do processamento dos fragmentos com novos vértices (quando necessários) e arestas, os quais são derivados da relação de referências não *containment* entre os elementos do modelo de entrada. Assim, todos os elementos do modelo de entrada são representados a partir de vértices, e as referências entre esses elementos e seus atributos, são estabelecidos por meio de arestas. Por fim, os vértices e arestas são estruturados em um GraphFrames.

Na Figura 4.7, há um exemplo com elementos do modelo `Families` traduzido em vértices (Sub figura 4.7(a)) e arestas (Sub figura 4.7(b) para um GraphFrames, o qual foi estruturado para a Dc4MT. Nesse exemplo, círculos e retângulos são usados para demonstrar a estrutura dos elementos e a relação entre eles no grafo. Por exemplo, os vértices e arestas sinalizados em vermelho indicam a estrutura do elemento `lastName Sailor` e os em azul indicam o elemento `firstName David`. A relação desses dois elementos está assinalada na aresta onde o valor da coluna `src` está circulado em vermelho e o valor da coluna `dst` está circulado em azul. A junção dessas estruturas (*match* entre `id`, `src` e `dst`) permite identificar que `David` é um filho (`sons`) e pertence à família `Sailor`.

Os elementos do modelo extraídos para um GraphFrames estão estruturados de modo que eles possam ser consultados e processados para diferentes propósitos. Por exemplo, identificar o grau[3] de vértices no grafo. Vértices com grau igual zero são vértices isolados e com grau igual 1 são vértices folhas (Mezic et al., 2019). A contagem de vértices e arestas e a visualização de grafos são meios que podem indicar a complexidade estrutural do modelo entrada. A densidade de referências entre os elementos do modelo de entrada é um aspecto que pode influenciar na escolha de uma estratégia de processamento de modelos. A densidade de referências refere-se a interconectividade entre os elementos de um modelo, que além da referência hierárquica incluem as referências entre elementos independente de suas posições na hierarquia (Burgueño, 2016). Por exemplo, os elementos de um modelo de Classes podem ser referenciados por um ou mais elementos de uma outra Classe. Essas classes podem estar contidas em Pacotes diferentes e essas referências podem ser estabelecidas por atributos como `super` e `type`.

[3] Medir a quantidade de arestas incidentes nos vértices (origem e destino). As funções *outDegrees()* e `inDegrees()` são utilizadas no GraphFrames para essa finalidade.

```
+-------+---------+
|id     |value    |
+-------+---------+
|1048592|sons     |
|1048590|sons     |
|1048582|Brandon  |
|1      |Family   |
|1048576|March    |
|0      |Family   |
|1048585|Sailor   |
|1048589|Jackie   |
|1048595|Kelly    |
|1048587|Peter    |
|1048591|David    |
    ....
```

(a) Vértices do GraphFrames

```
+-------+-------+---------+
|src    |dst    |key      |
+-------+-------+---------+
|1      |1048585|lastName |
|0      |1048576|lastName |
|1      |1048590|2        |
|1048583|1048584|firstName|
|1048592|1048593|firstName|
|1048594|1048595|firstName|
|1048577|1048578|firstName|
|1      |1048586|0        |
|1      |1048594|4        |
|1      |1048592|3        |
|1048590|1048591|firstName|
        ...
```

(b) Arestas do GraphFrames

Figura 4.7: Tradução de Elementos do Modelo Families para um GraphFrames

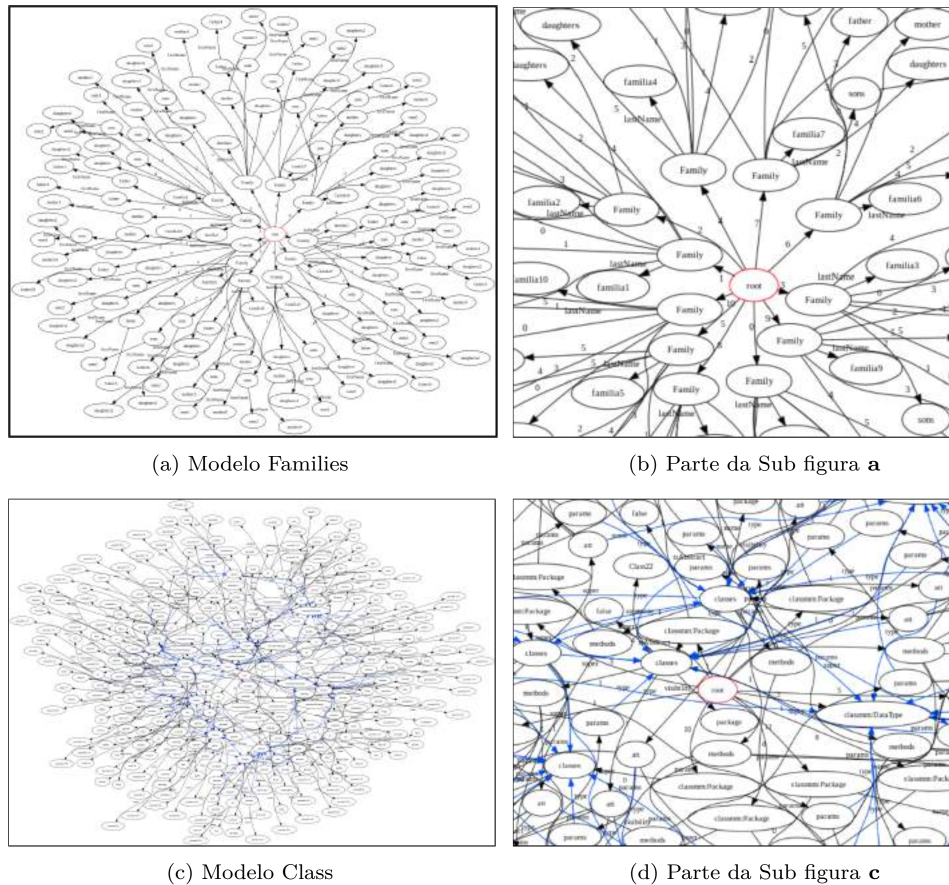
4.4.2 Visualização de Densidade de Referências

Na Figura 4.8, é mostrado um exemplo da visualização da densidade de referências e a interconectividade entre os elementos dos modelos Families e Class. Na Sub figura 4.8(a), o modelo Families contendo 10 famílias, em que cada família (pai e mãe) possuem dois filhos e duas filhas é apresentado em formato grafo. Um zoom centralizado dessa sub figura é mostrado na Sub figura 4.8(b) para uma melhor visualização. Esse procedimento (Sub figuras 4.8(c) e 4.8(d)) é aplicado para o modelo Class formado por 10 pacotes e 10 classes, cada classe é formada por 1 à 6 atributos e métodos. De acordo com as sub figuras contidas na Figura 4.8, nota-se que a densidade (o tamanho $\sum_{V+E}$ (V=vértices e E=arestas) de um grafo denso é proporcional a V^2) do grafo representado nas Sub figuras 4.8(a) e 4.8(b) é menor do que a do grafo das Sub figuras 4.8(c) e 4.8(d). Pode se dizer que o grafo formado pelo modelo Families é esparso (o tamanho de um grafo esparso é proporcional a V).

A visualização de modelos em formato grafo pode revelar a complexidade estrutural de cada modelo observado e expor a densidade de referências entre seus elementos. A visualização da densidade de referencias entre elementos está em destaque com arestas em azul a partir do vértice `root` em vermelho. Assim, observa-se que o grafo que representa o modelo Families não possui nenhuma aresta em azul. Isso significa que os elementos desse modelo estão conectados no elemento raiz (root) por meio dos respectivos elementos `Family` formando uma hierarquia. Ao contrário do modelo Class, não há nenhum outro tipo de referência (`ref/`) entre os elementos do modelo Families.

A implementação da visualização de modelos utiliza a API graphviz-java[4] e a interface da classe `Transformer` implementada no módulo Transformadores. Na próxima seção, uma especificação de transformação de modelos é apresentada.

[4]https://github.com/nidi3/graphviz-java



(a) Modelo Families

(b) Parte da Sub figura **a**

(c) Modelo Class

(d) Parte da Sub figura **c**

Figura 4.8: Modelos Families e Class Visualizados em Formato Grafo

4.5 ESPECIFICAÇÃO DE TRANSFORMAÇÕES DE MODELOS

As especificações de transformações de modelos são baseadas em junções (`join`), uma vez que os elementos do modelo de entrada estão estruturados em vértices e arestas, exigindo a junção desses elementos. O uso de junções está suscetível à dependência *Wide* entre partições de dados `parent-child` (Seção 4.2). Isso significa que a operação `shuffle` pode ser executada com frequência durante as transformações de modelos, principalmente para os modelos como Class, DBLP e Imdb (esse modelos são apresentados no próximo capítulo). A operação `shuffle` penaliza o desempenho das execuções paralelas/distribuídas. Para minimizar essa questão, foi adotada uma estratégia de redefinição do número de partições que são utilizadas pela operação `shuffle` do Spark. A quantidade de partições é atribuída diretamente na configuração do ambiente antes da execução, neste caso da TM. A aplicação dessa estratégia é apresentada no próximo capítulo.

A transformação de modelos como a `Families2Persons`, em que o modelo de entrada possui somente a referência hierárquica em um nível (Family-Members) os comandos `join` utilizados nos filtros geram dependências do tipo *Narrow*, já que os fragmentos desse modelo contêm famílias completas. Isso significa que no processo de Extração os elementos que compõem cada família tendem a ficar próximos. O Spark busca manter essa proximidade entre as partições `parent` e `child`, em que os `ids` dos elementos contidos no `gf.vertices` formam a referência entre os elementos. Esses ids (particionador RDD) são assinalados para o `gf.edges` para as colunas `src` e `dst`. No entanto, a execução dos comandos `join` utilizados na regra `Family2Person` (Listagem 4.4) que faz a junção

entre o sobrenome e o primeiro nome de cada membro de uma família, separando por gênero (Male ou Female) necessitará da operação `shuffle`. Visto que, a junção de partes de cada família requer que as partições em RDDs sejam combinadas a fim de satisfazer essas junções, formando dependências *Wide*.

Especificações de TMs em Dc4MT requerem que o modelo de entrada encontre-se traduzido para o GraphFrames em vértices e arestas. A especificação não leva em conta se a transformação será executada em modo local (paralelo) ou em um ambiente distribuído. No módulo Transformadores, a especificação é concretizada por meio da extensão da classe `Transformer` que permite especificar diferentes tipos de transformações. Por exemplo,

```
class Family2Person extends Transformer {
   override def transform(spark:  SparkSession,
   gf:  GraphFrame):  GraphFrame = {...}
```

As especificações de filtros (como *helpers* em ATL) e das regras de transformação são em estilo declarativo com chamadas para as funções fornecidas pela API GraphFrames. As regras em Dc4MT são baseadas no padrão *matching*, isso significa que a rastreabilidade é implícita e transiente entre os elementos dos modelos de entrada e saída.

O exemplo de transformação `Family2Person` é usado para ilustrar as especificações de filtros e de uma regra de transformação. O resultado da extração do modelo de entrada `Families` está contido em vértices (`gf.vertices`) e arestas (`gf.edges`), e uma vez atribuídos aos DataFrames `vertices` e `edges` podem ser manipulados sem a utilização do prefixo `gf`. Isso simplifica as chamadas de funções usadas nas expressões de filtros e regras de transformação. Na Listagem 4.3, três filtros são apresentados:

- o filtro na linha 1, seleciona todas as arestas em que o(s) valor(es) da coluna `key` coincide com o valor `"lastName"`;
- da mesma forma que o anterior, o filtro da linha 2 seleciona as arestas em que a coluna `key` possui valor igual `"firstName"`;
- o filtro nas linhas 3 e 4 seleciona os sobre nomes de cada família.

O resultado de cada filtro é atribuído aos respectivos DataFrames `lstNmFamily`, `fstNmFamily` e `lstNmFamilies`, os quais são utilizados em filtros subsequentes e na regra de transformação `Families2Persons` (Listagem 4.4). Os filtros nas linhas 1 e 2 (Listagem 4.3) são considerados simples, uma vez que eles utilizam apenas a função `filter` seguida por uma condição. O processamento desses filtros requer somente arestas. O filtro nas linhas 3 e 4 exige a junção do DataFrame `lstNmFamily` com os vértices (`vertices`) para obter o sobrenome de cada família. A junção (`join`) é do tipo `inner` e não precisa ser especificado na expressão, já que esse tipo é considerado padrão na API DataFrame. Esse filtro seleciona do DataFrame `lstNmFamily` (função `select`) as colunas `src` e `dst`, compondo o LHS (*Left-Hand Side*) da junção. A coluna `src` é utilizada para constituir o resultado final do filtro e a coluna `dst` para compor a expressão da junção ($ *"dst"=== $ "id"*) com a coluna `id` do vértice contido na GraphFrames (`vertices`). O RHS (*Right-Hand Side*) da junção inclui o `vertices` e o resultado final composto pelas colunas `src` e `value`.

Listagem 4.3: Filtros para os Elementos `lastName` e `firstName` do Modelo Famlies

```
1 val lstNmFamily = edges.filter("key = 'lastName'")
2 val fstNmFamily = edges.filter("key = 'firstName'")
3 val lstNmFamilies = lstNmFamily.select($"src", $"dst")
4   .join(vertices, $"dst" === $"id").select($"src", $"value")
5   ...
```

Os filtros são úteis para selecionar elementos de interesse do grafo e organizá-los em DataFrames, simplificando a especificação de regras de transformação. No entanto, a utilização de `joins` impõem desafios como a junção de tipos complexos, nome duplicado de colunas, diferentes tipos de expressões, entre outros. Em um ambiente de paralelismo implícito é necessário entender a(s) estratégia(s) de comunicação entre os nós adotada(s) por esse ambiente. O entendimento da estratégia de comunicação pode auxiliar na especificação de junções a fim de minimizar o impacto no desempenho das execuções de tais junções.

A regra de transformação `Families2Persons` é apresentada na Listagem 4.4. O elemento sobrenome de cada família (`.alias("lastName")`) e o índice desse elemento (`origem`) são selecionados do DataFrame `lstNmFamilies` como colunas LHS ao comando `join`. A coluna `src` do DataFrame `fstNmMale` é utilizada como coluna RHS para formar o *matching* entre os índices dos elementos sobrenome (`lastName`) e nome (`firstName`) com a seguinte expressão `.join(fstNmMale, $"src"===$"origem")`. Essa expressão assegura como resultado o nome e o sobrenome de cada família do gênero masculino. Desse resultado, são selecionadas as colunas `lastName` e `value`, uma vez que a coluna possui o elemento nome. Neste caso, o comando `select` é usado com a função `concat` para vincular a coluna `lastName` à coluna `value` formando a coluna `fullName`. O resultado dessa transformação é atribuído ao DataFrame `personMale`. O procedimento para que o DataFrame `personFemale` receba os respectivos elementos resultantes da transformação (linhas 6 à 9) é o mesmo descrito para o DataFrame `personMale`.

Listagem 4.4: Regra de Transformação `Families2Persons`

```
1 val personMale = lstNmFamilies
2   .select($"src".as("origem"), $"value".alias("lastName"))
3   .join(fstNmMale, $"src" === $"origem")
4   .select(concat($"lastName", lit(" "), $"value") as "fullName")
5
6 val personFemale = lstNmFamilies
7   .select($"src".as("origem"), $"value".alias("lastName"))
8   .join(fstNmFemale, $"src" === $"origem")
9   .select(concat($"lastName", lit(" "), $"value") as "fullName")
```

O resultado da regra de transformação `Families2Persons` está nos DataFrames `personMale` e `personFemale` que são agrupados no DataFrame `personsDF` pela função `union` com a coluna `Gender` para separar o gênero de cada pessoa. Na Listagem 4.5, são apresentadas as especificações dessa junção e a instancia dessa transformação. A função `coalesce(1)` interfere no particionamento de dados do framework Spark atribuindo uma única partição (parâmetro = 1) ao DataFrame `personsDF`. Desse modo, a saída da transformação é direcionada para um único arquivo (uma partição de saída)

Listagem 4.5: Especificação da Saída do Resultado da Transformação

```
1 val personsDF = personMale.withColumn("gender", lit("Male")
2   .union(personFemale.withColumn("gender",lit("Female")))
3   .coalesce(1)
```

O DataFrame `personsDF` é enviado ao nó Master que finaliza o processo de transformação instanciado a partição no sistema de arquivos. Para obter o formato da saída em XMI ou JSON a respectiva biblioteca é usada a partir do DataFrame `personsDF`. Na listagem 4.6 é mostrado um exemplo do modelo de saída no formato XMI como o resultado final da operação de transformação. A escrita (`write`) é uma operação de Ação no framework Spark. Essa operação dispara a computação contida no plano de execução (DGA) construído pelas operações de Transformação (*lazy evaluate*), neste caso formado pelos comandos (`join`, `filter`, `union`, ...) utilizados nos filtros e nas regras de transformação.

Listagem 4.6: Parte do Modelo de Saída Persons

```
<Persons>
  <gender> Male </gender>
   <fullName> March Jim </fullName>
  <gender> Male </gender>
   <fullName> Sailor Dylan </fullName>
  <gender> Female </gender>
   <fullName> March Cindy </fullName>
  <gender> Female </gender>
   <fullName> March Brenda </fullName>
      ...
</Persons>
```

4.6 CONSIDERAÇÕES FINAIS

Neste capítulo, a abordagem Dc4MT foi descrita, iniciando com uma visão geral dos módulos Fragmentador, Extrator e Transformadores, seguida pela representação dos passos que envolvem a execução desses módulos. O modelo de comunicação do ambiente de execução utilizado na implementação da Dc4MT foi descrito com a intenção de mostrar a dependência de dados na troca de mensagens entre os nós durante processamento paralelo/distribuído de dados. Observou-se que a dependência de dados está associada ao paralelismo implícito e pode ter influência sobre o desempenho das operações de Transformação e Ação realizadas no ambiente de execução. A execução do módulo Fragmentador não é afetada pelo paralelismo implícito, uma vez que esse módulo é executado no nó Master. Já os módulos Extrator e Transformadores são afetados em decorrência da troca de mensagens entre os nós durante a execução paralela/distribuída.

Os módulos da Dc4MT foram caracterizados com exemplos e representações por meio de modelos e pseudocódigos (algoritmos). Uma estratégia para a fragmentação de modelos foi apresentada, assim como a visualização de modelos em modo grafo. A visualização teve o intuito de mostrar a complexidade do modelo sob a perspectiva de densidade de referências e complexidade estrutural do modelo visualizado. A resolução de referências entre os elementos do modelo de entrada foi apresentada como parte do módulo Extrator. Por fim, a especificação de um exemplo de transformação de modelos foi descrita, utilizando o modelo Families como modelo de entrada na transformação `Families2Persons`. Esse exemplo de transformação foi usado como um *baseline* na apresentação da abordagem Dc4MT. Os aspectos técnicos da Dc4MT também foram descritos pautados no ambiente de execução do framework Spark e nas suas APIs RDD, GraphFrames e DataFrame, no entanto a Dc4MT pode ser implementada para qualquer framework de paralelismo implícito. No próximo capítulo, são apresentados os resultados com os experimentos da implementação descrita neste capítulo.

5 EXPERIMENTOS

Neste capítulo, a implementação da Dc4MT é experimentada sob duas perspectivas: em modo local e em modo distribuído. Em modo local, são executadas as operações de Fragmentação, Extração e Transformação, sendo que essas operações são executadas em modo paralelo, exceto a Fragmentação que é executada em modo sequencial. As execuções em modo local buscam medir o desempenho dessas operações. As execuções em modo distribuído visam a escalabilidade da operação Transformação. Em ambas, os experimentos buscam a factibilidade, escalabilidade e desempenho de abordagem Dc4MT. Uma única implementação da Dc4MT na linguagem Scala sob o framework Spark é adotada (código fonte[1]) para as execuções paralelas e distribuídas.

Um conjunto de (*datasets*) modelos em formato XMI e JSON é utilizado. Os modelos em JSON são usados para comparar a estratégia de extração entre os formatos XMI e JSON (Tabela 5.1). Já os modelos em XMI são usados para validar o desempenho das operações de fragmentação, extração e transformação. Para a serialização dos modelos de entrada nos formatos XMI e JSON, as respectivas bibliotecas `javax.xml.streaming`[2] e `Jackson`[3] são utilizadas. Para garantir esses formatos aos modelos de saída, a biblioteca `com.databricks`[4] é utilizada na operação de escrita desses modelos. Na Tabela 5.1, são apresentados quatro tipos de modelos utilizados na experimentação da Dc4MT.

Os *Datasets* do tipo Family são modelos formados por elementos que representam uma família com paí, mãe, filhas e filhos. Esses modelos (Tabela 5.2) foram gerados (Origem) por meio de uma aplicação escrita em Scala. Os modelos do tipo Class (Diagrama de Classes em formato XMI) foram obtidos do Atenea MTBenchmark[5]. Os *Datasets* DBLP (*Digital Bibliography Library Project*) e Imdb (*Internet Movie Database*) são extraídos do Projeto Lintra[6]. O DBLP é um banco de dados bibliográfico na área da ciência da computação. Esse banco provê dados bibliográficos sobre os principais periódicos e *proceedings* relacionados à ciência da computação[7]. O Imdb é um banco de dados online sobre filmes, vídeos e programas de TV que contêm informações a respeito de elenco, produção, avaliações, entre outras[8].

A densidade de referências estabelecida para os modelos Class está relacionada à presença de elementos como classes, atributos e métodos, exceto para o modelo `Class-10`^{5}`-0` que não possui referências entre elementos além da hierarquia entre os elementos Pacote e Classes (Tabela 5.3). Por exemplo, o modelo `Class-10`^{5}`-1` significa que ele é tem 1.000.000 classes (10^5 representa 10 + 5 zeros) com uma densidade de elementos igual 1 (10^5-**1**). As classes desse modelo são dividas entre 10 pacotes, de modo que cada pacote contenha 100.000 classes como elementos filho. Cada elemento classe tem um atributo e/ou um método, formando uma hierarquia entre os elementos pacote, classe, atributo e método. Os elementos atributo e método podem ser considerados como elementos folha

[1] https://github.com/lzcamargo/Dc4MT
[2] https://docs.oracle.com/javase/8/docs/api/index.html?javax/xml/stream/package-summary.html
[3] https://github.com/FasterXML/jackson
[4] https://github.com/databricks/spark-xml
[5] `wget` http://atenea.lcc.uma.es/Descargas/MTBenchmark/classModels/model-100000-1.xmi
[6] http://atenea.lcc.uma.es/projects/LinTra.html
[7] https://dblp.uni-trier.de/
[8] https://www.imdb.com/ O IMDb foi criado nos anos 90 como um banco de dados de filmes por grupos de usuários da Usenet. Atualmente a IMDb é uma subsidiária Amazon

da hierarquia. Além disso, esses elementos podem ter referências e ou serem referenciados independentemente de sua posição na hierarquia (densidade de referências). Essa estrutura se repete para os demais modelos do tipo Class (`Class-10`^{5}`-1, Class-10`^{5}`-2, ...`). Na medida em que a presença dos elementos atributos e métodos aumenta (1 a 6) nos respectivos modelos, a densidade de referências é expandida. Na coluna Densidade Ref. da Tabela 5.1 são classificados os tipos de modelos com a referência hierárquica (hier.) e/ou tipos de modelos com as referências entre elementos (entreElemen.), o Formato do tipo de modelo e a sua Versão.

Tabela 5.1: Tipos de Dataset Utilizados nos Experimentos

Tipo Dataset	**Origem**	**Densidade Ref.**	**Formato e Versão**
Family	gerado	hierárquica.	XMI v2.0 e JSON
Class	download	hier. e entreElem.	XMI v1.0
DBLP	download	hier. e entreElem.	XMI v1.0
Imdb	download	hier. e entreElem.	XMI v1.0

Os modelos do tipo Family possuem densidade de referência hierárquica, em que os elementos Father, Mother, Sons e Daughters possuem o atributo `FirstName` e estão agregados ao elemento `LastName`, formando uma hierarquia. Os sufixos atribuídos ao nome dos modelos do tipo Family, indicam a formação estrutural do modelo. Por exemplo, o modelo Families-11^3-8 é formado por 11000 famílias (11^3) e cada família é constituída por 8 membros: pai, mãe, três filhos e três filhas. Já o Families-11^6-4 tem 11000000 de famílias e cada família possui pai, mãe, um filho e uma filha (4 membros). A versão 2.0 (v2.0) do formato XMI para os modelos do tipo Family é porque esses modelos foram gerados por uma aplicação. Os demais modelos com a versão 1.0 para o formato XMI foram obtidos da Internet.

Os diferentes tipos e tamanhos de modelos utilizados nos experimentos visam a diversificação do conjunto de regras de transformações e a ampliação do escopo de validação para a Dc4MT. Na próxima seção, os resultados dos experimentos com esses modelos são apresentados sob a perspectiva de execução paralela. O tempo de cada execução é mensurado em segundos por meio da função `System.currentTimeMillis(/1000.000)`. Os experimentos foram realizados em um notebook com o sistema operacional Ubuntu 19.10 de 64 Bits contendo 15,4 GB de memória RAM, um SSD ADATA_SU810NS38_SATA_128_GB, um processador (i7-8565U) de 1.80GHz com 4 núcleos e duas *threads* por núcleo (8 CPUs). A IDE (*Integrated Development Environment*) IntelliJ IDEA Community Edition[9] foi adotada como ambiente de desenvolvimento e execução. O parâmetro `-Xmx12G` foi atribuído à JVM através da IDE para melhor alocação memória. Integrados a esse ambiente estão a Linguagem Scala 2.11.0[10], JRE 1.8[11] (*Java Runtime Environment*), Spark 2.4.3 *pre-built for* Apache Hadoop 2.7[12] e a ferramenta SBT[13] (*Scala Build Tool*) para o controle de dependência entre pacotes e bibliotecas como a `com.databricks`, `javax.xml.streaming`, `Jackson`, entre outras.

[9]https://www.jetbrains.com/idea/
[10]https://docs.scala-lang.org/
[11]https://www.java.com/en/
[12]https://spark.apache.org/download
[13]https://www.scala-sbt.org

5.1 EXTRAÇÃO PARALELA DE MODELOS

Os resultados apresentados nesta seção foram obtidos das 55 execuções, envolvendo a fragmentação de modelos, extração e a contagem de vértices e arestas. Os experimentos com a extração de modelos visam mostrar o desempenho dessas operações, como também identificar o tamanho e o tipo de grafo derivados da extração de modelos. Na Figura 5.1, é exibido o tempo de execução da fragmentação de cada modelo. Esse resultado é extraído das Tabelas 5.2, 5.3, 5.4 e da fragmentação do modelo DBLP. Como a fragmentação é executada no nó Master, a configuração do ambiente de execução é irrelevante em relação ao número de nós. A densidade de referências não impacta no desempenho da execução da Fragmentação, uma vez que a estratégia de fragmentação preserva as referências entre os elementos. Como pode ser visto na Figura 5.1, o tempo de execução aumenta na medida em que o tamanho do modelo também aumenta, influenciando no desempenho da fragmentação.

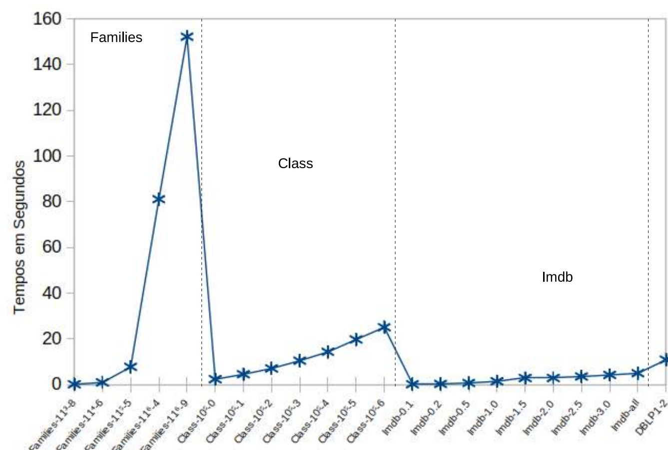


Figura 5.1: Resultado das Fragmentações de Modelos

O tempo de execução de cada operação de Fragmentação dos modelos está contido nas Tabelas 5.2, 5.3 e 5.4. Nessas tabelas também estão os resultados da contagem de vértices e arestas dos modelos, exceto para o modelo `Families-11`^{7}`-4`, esse modelo não é executado devido a limitação do ambiente local. A contagem é obtida após a extração dos modelos por meio da função `count()` presente no Spark (e.g., `gf.vertices.count()`). Ao observar a quantidade de vértices e arestas dos modelos do tipo Family presentes na Tabela 5.2, nota-se que há uma linearidade em relação a quantidade de vértices e arestas, uma vez que a quantidade de vértices menos 1 (V -1) é igual a quantidade de Arestas E ($\sum_{V-1} = \sum_{E}$). Isso significa que os grafos derivados da extração dos modelos do tipo `Family` são completos[14] e simplificados. Além disso, os resultados dos somatórios de vértices e arestas também revelam que os modelos possuem apenas referência hierárquica entre seus elementos.

Um grafo completo e simples é o resultado da extração do modelo `Class-10`^{5}`-0`, uma vez que esse modelo possui densidade de referência igual a zero. Os demais modelos do tipo `Class` contidos na Tabela 5.3 formam grafos completos, mas não simples[15], visto que a quantidade de arestas é maior do que a quantidade de vértices. Isso é devido a densidade (1 a 6) de referências contidas nesses modelos.

[14] Um grafo é completo se todo par ordenado de vértices distintos forma um arco (Mezic et al., 2019)
[15] Um grafo é considerado simples quando não contém *loops* ou múltiplas arestas (Mezic et al., 2019).

Tabela 5.2: Extração dos Modelos Família para GraphFrames

Especificação dos Datasets			**Execução Families2GraphFrame**		
Dataset	**Qtde Elem.**	**Tamanho**	**Fragmentação**	**Vértices**	**Arestas**
Families-11^3-8	88000	3,1 MB	0.65s	180019	180018
Families-11^4-6	660000	23.7 MB	0.96s	1400015	1400014
Families-11^5-5	5500000	221,7 MB	7.75s	12000013	12000012
Families-11^6-4	44000000	1.9 GB	81.10s	100000011	100000010
Families-11^6-9	99000000	3.6 GB	152.15s	200000021	200000020
Families-11^7-4	440000000	19.0 GB	ne	ne	ne

ne não executada em decorrência da limitação do ambiente.

Tabela 5.3: Resultado da Extração dos Modelos Class para o GraphFrames

Especificação dos Datasets			**Execução Class2GraphFrames**		
Dataset	**Qtde Elem.**	**Tamanho**	**Fragmentação**	**Vértices**	**Arestas**
Class-10^5-0	1000010	80 MB	2.37s	4000027	4000026
Class-10^5-1	30000010	170 MB	4.53s	7501357	8751829
Class-10^5-2	50000010	283 MB	7.08s	11306915	14210541
Class-10^5-3	70000010	420 MB	10.46s	14521411	19711435
Class-10^5-4	90000010	583 MB	14.32s	16790460	25781832
Class-10^5-5	110000010	773 MB	19.73s	18068614	32629082
Class-10^5-6	130000010	1.1 GB	25.08s	18596488	40771356

Na Tabela 5.4, são apresentados os tempos da fragmentação (Fragmentação) dos modelos tipo Imdb e da tradução desses fragmentos para o GraphFrames (`Imdb2GF`). Os grafos gerados na tradução dos fragmentos é do tipo completo e não simplificado, já que os modelos Imdb possuem densidade de referências. Nota-se que o tempo de tradução dos modelos Imdb está entre `1,218s` a `1,852s`, uma variação muito pequena de tempo em relação ao tamanho dos modelos. Isto é devido ao efeito *lazy evaluate* na formação do grafo por meio da API GraphFrames, em que a computação de construção do grafo é disparada somente quando alguma operação de Ação é invocada. Antes disso, os comandos utilizados na operação de extração formam um plano lógico de execução. Quando uma operação de ação é executada, toda a computação contida no plano lógico de execução é transformada em um plano físico de execução e em seguida é executado. O Spark busca uma estratégia que privilegia o desempenho de execuções em um plano físico.

Tabela 5.4: Extração dos Datasets IMDB para GraphFrames

Dataset	**Tamanho**	**Elementos**	**Fragmentação**	**Imdb2GF**
Imdb-0.1	7.1 MB	100024	0,155s	1,218s
Imdb-0.2	14.3 MB	200319	0,293s	1,236s
Imdb-0.5	37.7 MB	500716	0,732s	1,336s
Imdb-1.0	81.9 MB	1013510	1,45s	1,417s
Imdb-1.5	129.7 MB	1511287	2.088s	1,486s
Imdb-2.0	181.9 MB	2019707	2,896s	1,579s
Imdb-2.5	232.6 MB	2509987	3,543s	1,678s
Imdb-3.0	287.9 MB	3017435	4,23s	1,774s
Imdb-all	345.4 MB	3531618	4,976s	1,852s

Os modelos Families expressados nos formatos XMI e JSON são utilizados como entrada nas execuções da operação de Extração, visando uma comparação entre esses formatos. Esses modelos possuem tamanho e quantidade de elementos iguais. Na Tabela 5.5, é mostrado o resultado dessas execuções, em que os modelos Families são traduzidos para grafos representados em GraphFrames (Families2GraphFrames). A intenção não é comparar as APIs `javax.xml.streaming` e `Jackson`, mas observar o desempenho da extração nesses dois formatos adotados pela Dc4MT. Os resultados mostrados na

Tabela 5.5 ratificam as afirmações de Smolders (2017) e Hili (2016), em que a serialização de modelos expressados no formato JSON tendem a apresentar um melhor desempenho quando comparado com modelos em XMI. O desempenho médio (cálculo de média simples da diferença entre os tempos de execuções XMI e JSON) das execuções de extração de modelos em JSON é superior 63%. Neste caso, essa diferença não se pode atribuir apenas ao formato, mas a outras questões como o processamento *streaming* para a JVM e as características específicas de cada API.

Tabela 5.5: Extração dos Modelos Families em XMI e JSON

Dataset	**XMI**	**JSON**
	Extração	**Extração**
Families-11^3-8	0.65s	0,38s
Families-11^4-6	0.96s	0,63s
Families-11^5-5	7.75s	2,40s
Families-11^6-4	81.1s	23,25s
Families-11^6-9	152.17s	45,59s

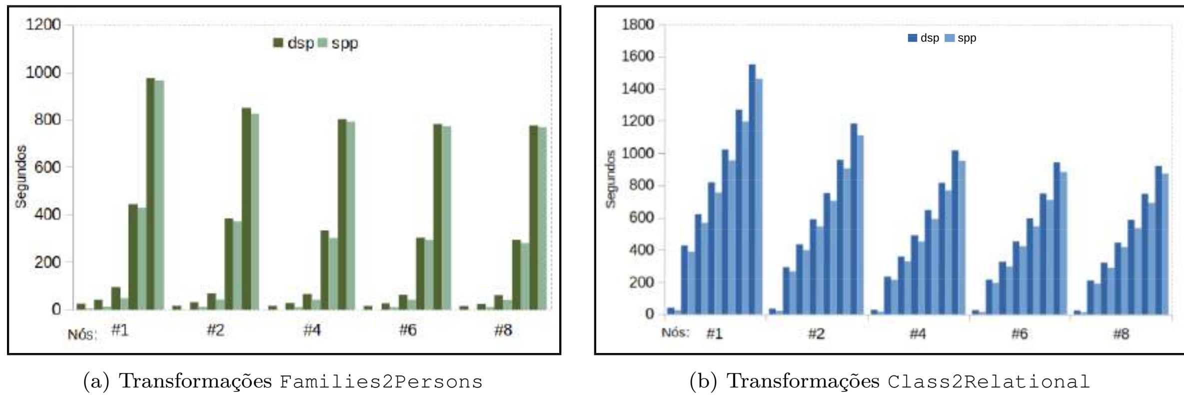
Qualquer execução de uma ou mais operações requer a configuração do ambiente de execução do Spark, em que parâmetros são atribuídos a esse ambiente. Nas execuções em paralelo, a interface `mt-spark.conf` (um arquivo manipulado via IDE) possibilita a atribuição de parâmetros ao Spark contexto. O Spark contexto (`SparkContext`) é o principal ponto de entrada para o acesso às funcionalidades do Spark e a representação de conexão aos nós do ambiente. São exemplos de parâmetros de configuração:

```
spark.master local[2]
spark.conf.set("spark.sql.shuffle.partitions", 600)
splicedxmlinjector.modelpath ../modelsExp/dblp12.xmi
```

A configuração do ambiente (`mt-spark.conf`) possibilita a combinação das seguintes execuções: somente a fragmentação de modelos (execução do módulo Fragmentador), extração de modelos (execução dos módulos Fragmentador e Extrator) e a transformação de modelos (execução dos módulos Fragmentador, Extrator e Transformadores). Na próxima seção, os resultados com experimentos de transformação de modelos em modo paralelo, são apresentados.

5.2 TRANSFORMAÇÕES PARALELA DE MODELOS

Os resultados apresentados nas próximas seções foram obtidos das 1760 execuções das transformações paralela de modelos em modo local. Essas execuções buscam mostrar o desempenho e a factibilidade das transformações paralelas de modelos em um ambiente de memória compartilhada. Na Figura 5.2, é apresentado o resultado dos experimentos relacionados ao desempenho das execuções das transformações dos modelos Families e Class. O desempenho é o tempo mensurado em segundos da fragmentação, extração e transformação desses modelos. O resultado das execuções em cada nó é exibido do modelo menor para o maior (e.g., Class-10^5-0 ao Class-10^5-6). Nos experimentos a quantidade de nós é atribuída via o ambiente de configuração (`mt-spark.conf`) e são executados no ambiente descrito na Seção 5.1 da seguinte forma: execuções em um único nó #1, #2, #4, #6 e em #8 nós. Cada modelo de entrada é executado oito vezes, sendo que as três primeiras execuções são descartadas devido a fase de aquecimento (*warm up*) da máquina virtual (JVM). Assim, o tempo de execução para cada modelo é obtido com o cálculo do tempo médio (média aritmética simples) das cinco execuções restantes. Nesse procedimento inclui as execuções que contêm o parâmetro de particionamento da operação



(a) Transformações Families2Persons (b) Transformações Class2Relational

Figura 5.2: Transformações dos Modelos Families e Class

shuffle alterado pelo ambiente de configuração (spp - *shuffle partition parameter*), como também as execuções sem alteração (dsp - *default shuffle partition*) do valor padrão (200 partições). O impacto da alteração do particionamento padrão é discutido adiante.

Os tempos de execução em um único nó quando comparados com as execuções em dois nós decrescem em média 23% para os modelos Families e 26% para os modelos Class. Esses percentuais são obtidos com um cálculo da porcentagem entre os tempos de execuções contidos nas colunas #1 e #2 das Tabelas 5.6 e 5.7. No entanto, esse declínio de tempo nas execuções não é o mesmo na medida em que novos nós (#4, #6, #8) são acrescidos ao ambiente de execução. Embora a inserção de nós no ambiente aumenta o paralelismo, a quantidade de memória é a mesma. Isso explica o baixo decréscimo de tempo entre as execuções a partir do nó #2, sobretudo para as transformações Families2Persons (próximo de uma linearidade).

Os modelos Families e Class representam dois tipos diferentes de modelos no que diz respeito a complexidade de processamento. Em um *benchmark* realizado por Benelallam et al. (2014), em que um conjunto de modelos utilizados no projeto Mondo[16] é classificado em termos de computação (*query*/transformação) e complexidade computacional (alta/baixa). Baseado nessa classificação e considerando as características dos modelos utilizados nos experimentos da Dc4MT, o termo adotado para classificar esses modelos é complexidade estrutural ao invés de complexidade computacional. Todos os modelos transformados pela implementação da Dc4MT são submetidos a dois tipos de computação *query* (filtros) e transformação (regras). Os modelos Families podem ser classificados como modelos de complexidade estrutural baixa, enquanto que os modelos Class são classificados com complexidade estrutural alta (exceto para o modelo Class−10^5−0 que possui somente uma hierarquia entre os elementos Package e Classes), assim como os modelos Imdb e DBLP. A visualização de trechos dos modelos Families e Class em modo grafo apresentada na Figura 4.8 (Seção 4.1) corrobora com essa classificação.

Nas Sub figuras a e b (Figura 5.2) pode se observar que a complexidade estrutural e o domínio de transformação exercem mais influencia no desempenho das execuções do que o tamanho do modelo, tendo em vista a capacidade do ambiente local de execução. Nas transformações dos modelos do tipo Family, o desempenho é degradado (aumento do tempo execução) pela limitação de memória do ambiente de execução na medida em que modelos maiores são submetidos à execução, neste caso a partir de 1.9 GB. O modelo Families−11^7−4 não foi submetido à execução (ne) devido à limitação do ambiente em relação ao tamanho desse modelo (19 GB). O tamanho dos modelos Class−10^5−0 e

[16]http://www.mondo-project.org/

`Families-11`^{4}`-6` são diferentes (respectivamente 80 e 27 MB) e possuem complexidade computacional baixa. Conforme as Tabelas 5.6 e 5.7, os tempos de execuções desses modelos são próximos devido ao domínio de transformação. A transformação do modelo *Class*-10^5-*0* exige uma especificação mais simplificada (menos instruções declarativas e *joins*) do que a transformação do modelo `Families-11`^{4}`-6`. O modelo *Class*-10^5-*0* é formado pelos elementos Package e Classe e seus atributos Nome. Esses elementos são transformados nos elementos Schema e Table. Além disso, há somente a hierarquia entre os elementos Package e Class, ao contrário do modelo `Families-11`^{4}`-6` que possui hierarquias entre os elementos Sobre nome e os membros da família Father, Mother, Sons e Daugthers que são transformados em pessoas.

Tabela 5.6: Resultado das Transformações Paralelas - Families2Persons

Datasets	**Execução Local – Quantidade de Nós**					
	SP	**#1**	**#2**	**#4**	**#6**	**#8**
Families-11^3-8	`dsp`	24s	16s	15,6s	15s	14,4s
	`spp=5`	5s	4,5s	4,5s	4,5s	4,5s
Families-11^4-6	`dsp`	41s	30s	26s	25s	23s
	`spp=5`	11s	10s	9s	9s	9s
Families-11^5-5	`dsp`	94s	68s	66s	63s	61s
	`spp=100`	49s	42s	41s	41s	40s
Families-11^6-4	`dsp`	445s	386	333s	302s	295s
	`spp=600`	427s	372s	301s	294s	281s
Families-11^6-9	`dsp`	976s	851s	804s	786s	776s
	`spp=700`	964s	826s	792s	773s	768s
Families-11^7-4	ne	ne	ne	ne	ne	ne

ne não executado. `dsp=200` partições, `spp=5, ..., 700` partições

O tamanho total dos modelos Families executados é 41% maior quando comparado ao tamanho total dos modelos Class (cálculo obtido da Coluna Tamanho incluída nas Tabelas 5.2 e 5.3). No entanto, o tempo de execução dos modelos Class é cerca de 72% superior quando comparado com o tempo de execução dos modelos Families (essa porcentagem é calculada dos valores em segundos contidos na Coluna #1 das Tabelas 5.6 e 5.7). Esse resultado indica que a complexidade estrutural e o domínio de transformação são aspectos que devem ser observados, uma vez que esses aspectos interferem no desempenho das transformações paralelas de modelos. O domínio das transformações `Class2Relational` é obtido do cenário *ClasstoRelational* da atlTransformations[17] constituído por sete regras, as quais possuem um conjunto de filtros. Além disso, inclui nesse domínio a junção dos resultados das regras para formar o modelo de saída.

Na Coluna SP (*Shuffle Partition*) das Tabelas 5.6, 5.7, 5.9 e 5.10 há linhas com a sigla `dsp`, indicando que o resultado das execuções não teve interferência no particionamento padrão da operação Shuffle (`dsp=200`). Por outro lado, nessa mesma coluna há valores inteiros atribuídos ao `spp` em um intervalo de 5 à 700 (e.g. `spp=5`), mostrando a intervenção no parâmetro `shuffle.partitions` do framework Spark (`spp`), cujo resultado das execuções com essas intervenções está nas linhas dessa coluna para cada modelo. Pode-se observar nessas tabelas que a intervenção aumenta o desempenho das execuções em relação as execuções que utilizam o particionamento padrão pela operação *Shuffle*. O impacto da interferência no `shuffle partitions` do Spark está detalhado na Tabela 5.8. Os valores são atribuídos ao parâmetro `shuffle.partitions` de acordo com a estratégia descrita na Seção 4.5. Por exemplo, modelos com o tamanho $<$ 100 MB o

[17]https://www.eclipse.org/atl/atlTransformations/#Class2Relational

Tabela 5.7: Transformações Class2Relational

Datasets	**Execução Local – Quantidade de Nós**					
	SP	**#1**	**#2**	**#4**	**#6**	**#8**
Class-10^5-0	dsp	41s	35s	27s	25s	24s
	spp=5	23s	20s	17s	16s	15s
Class-10^5-1	dsp	427s	292s	235s	214	210
	spp=100	387s	267s	213s	195s	191s
Class-10^5-2	dsp	621s	435s	359s	327s	320s
	spp=150	568s	399s	329s	297s	289s
Class-10^5-3	dsp	819s	590s	491s	452s	445s
	spp=300	756s	545s	451s	423s	418s
Class-10^5-4	dsp	1025s	753s	647s	598s	587s
	spp=400	957s	703s	593s	546s	538s
Class-10^5-5	dsp	1270s	961s	817s	751s	749s
	spp=500	1195s	905s	769s	709s	694s
Class-10^5-6	dsp	1549s	1186s	1019s	944s	923s
	spp=600	1461s	1112s	955s	885s	874s

dsp=200 partições ssp=5, ..., 600 partições

spp foi parametrizado com valor igual 5, uma vez que o ambiente de execução é composto por 4 nós físicos ($N_{no} + 1$).

Os percentuais apresentados na Tabela 5.8 são os percentuais médios ($\overline{P}$) obtidos com a totalização dos valores das execuções com spp (t_{spp}) sobre a totalização das execuções com parâmetro padrão dsp (t_{dsp}). O cálculo $\overline{P} = \frac{t_{spp}}{t_{dsp}}$ utiliza os tempos das execuções de todos os modelos. A influência no desempenho das execuções em percentuais mostrada na Tabela 5.8 pelo spp estão organizados sob três perspectivas:

- spp < 200 partições, parametrização do shuffle partitions menor que o particionamento padrão. Essa parametrização é adotada para modelos com tamanho menor 500 MB. Neste caso, o desempenho em média é de 33% superior em relação ao dsp;

- spp > 200 partições, parametrização maior que a padrão utilizada para modelos, cujo tamanho é superior a 500 MB. O desempenho é de 9% maior em relação dsp;

- Média-Modelos, nessa perspectiva busca-se visualizar a média dos tempos de execuções para cada tipo de modelo. A média geral dos tempos das execuções paralela em modo local é de 19%.

As porcentagens dos itens acima são calculadas como uma média dos valores contidos em cada linha da Tabela 5.8, exceto para as colunas com valor igual 0% em que o parâmetro spp não foi usado.

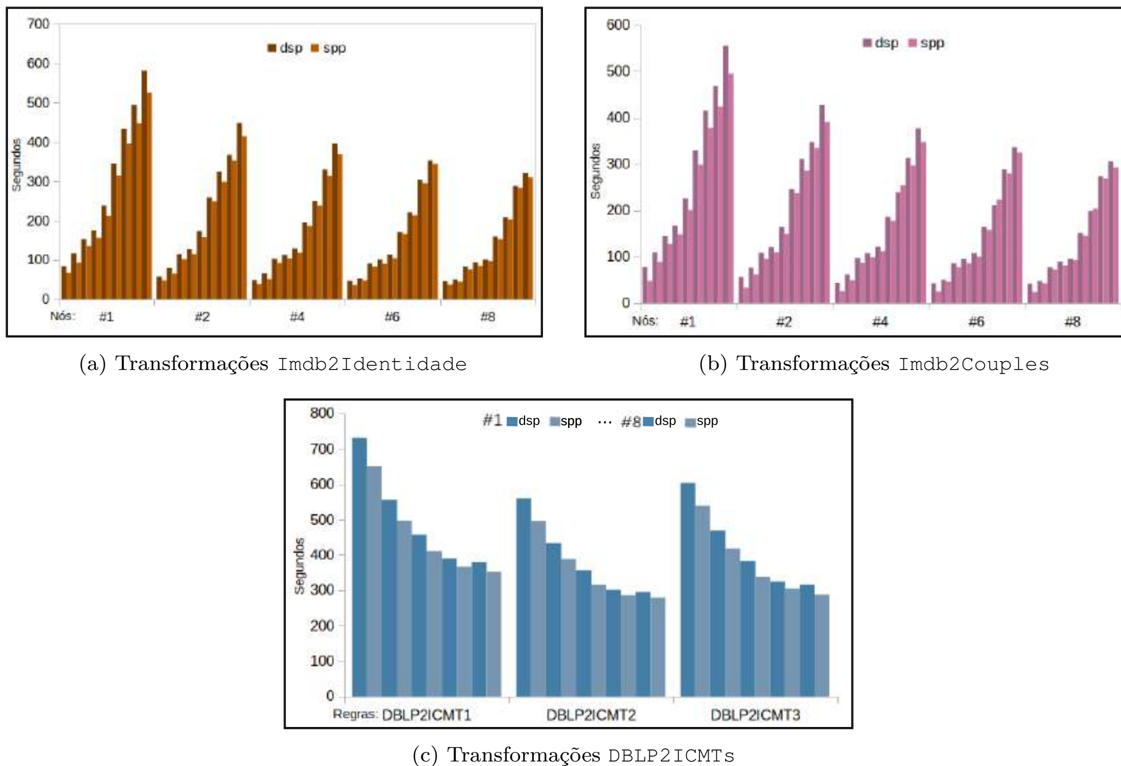
Tabela 5.8: Impacto do Shuffle Partitions nas Execuções Paralelas das Transformações

spp/dsp	**Families**	**Class**	**Imdb**	**DBLP**
spp $<$ 200	67%	20%	12%	0%
spp $>$ 200	4%	13%	0%	10%
Média-Modelos	42%	12%	12%	10%

De acordo com a Tabela 5.8, nota-se que a parametrização spp menor que 200 apresenta um desempenho em média três vezes superior quando comparado com o desempenho médio, em que a parametrização é maior que 200. Isso é decorrente do tamanho do modelo, da baixa complexidade estrutural e do domínio de transformação. O

desempenho das execuções dos modelos Class, Imdb e DBLP é aproximado, em média 11% ((12 + 12 + 10)/3), mas inferior quando comparado ao desempenho das execuções dos modelos Families, principalmente com a parametrização `spp < 200`. Por outro lado, quando o tamanho do modelo (`spp > 200`) afeta a disponibilidade de recursos do ambiente, o desempenho diminui, mas ainda é superior (`4%`) em relação às execuções com `dsp` (*shuffle partition* padrão). Porém, não foi incluído nos experimentos a verificação de que, em que ponto a interferência no particionamento do *shuffle partition* do ambiente favorece o desempenho das execuções.

Na Figura 5.3, é mostrado o resultado das transformações dos modelos Imdb e DBLP. Nas Sub figuras 5.3(a) e 5.3(b) são apresentados os resultados das transformações `Imdb2Identidade` e `Imdb2Couples`. A transformação `Imdb2Identidade` busca percorrer todos os elementos do modelo Imdb e copiá-los para o mesmo repositório de entrada. Já `Imdb2Couples` busca encontrar pares de pessoas que tenham trabalhado juntos em ao menos três filmes. Na Sub figura 5.3(c) são mostrados os tempos de execuções de três regras de transformações `DBLP2ICMT1`, `DBLP2ICMT2` e `DBLP2ICMT3`. Os tempos de execuções dessas regras são apresentados do nó #1 ao nó #8 (da esquerda para direita).



(a) Transformações `Imdb2Identidade`

(b) Transformações `Imdb2Couples`

(c) Transformações `DBLP2ICMTs`

Figura 5.3: Transformações dos Modelos `Imdb` e `DBLP`

Os valores em segundos que compõem as Sub figuras 5.3(a), 5.3(b) e 5.3(c) estão contidos nas Tabelas 5.9 e 5.10. Na Tabela 5.9, estão os resultados das execuções das regras `Imdb2Identidade` e `Imdb2Couples`, utilizando um conjunto de diferentes tamanhos do modelo Imdb como entrada. Nesse conjunto, nenhum modelo possui tamanho superior à 500 MB, em decorrência disso os valores adotados para o parâmetro `spp` são inferiores à 200.

Na Tabela 5.10, são exibidos os resultados das execuções de três regras de transformações, as quais utilizam um modelo DBLP com o tamanho superior à 500 MB, nesse caso o valor atribuído ao parâmetro `spp` é igual a 600. A variação dos tempos de execuções

Tabela 5.9: Transformações de Modelos Imdb

Datasets	SP	IMDb2identidade - Nós					FindCouple - Nós				
		#1	#2	#4	#6	#8	#1	#2	#4	#6	#8
Imdb-0.1	dsp	84s	58s	49s	47s	46s	79s	57s	44s	43s	42s
	spp=5	68s	48s	39s	36s	37s	48s	34s	27s	26s	25s
Imdb-0.2	dsp	116s	81s	66s	54s	51s	110s	77s	62s	51s	48s
	spp=5	93s	65s	52s	48s	45s	89s	62s	50s	46s	43s
Imdb-0.5	dsp	153s	115s	103s	91s	83s	144s	109s	97s	86s	78s
	spp=5	136s	102s	92s	83s	76s	128s	96s	87s	78s	72s
Imdb-1.0	dsp	175s	127s	113s	101s	94s	167s	121s	108s	96s	90s
	spp=10	156s	115s	104s	90s	85s	149s	110s	99s	86s	81s
Imdb-1.5	dsp	238s	174s	129s	114s	101s	226s	164s	122s	108s	96s
	spp=30	212s	158s	118s	105s	98s	201s	150s	112s	100s	93s
Imdb-2.0	dsp	346s	259s	196s	172s	160s	329s	246s	186s	164s	152s
	spp=40	315s	249s	187s	166s	153s	299s	237s	178s	158s	145s
Imdb-2.5	dsp	434s	325s	250s	221s	208s	415s	311s	239s	211s	199s
	spp=60	396s	299s	267s	234s	213s	378s	286s	255s	224s	204s
Imdb-3.0	dsp	494s	367s	330s	304s	289s	468s	348s	313s	288s	274s
	spp=80	447s	353s	314s	295s	283s	424s	334s	297s	280s	268s
Imdb-all	dsp	582s	448s	396s	353s	321s	555s	427s	377s	336s	306s
	spp=100	526s	415s	369s	345s	310s	495s	391s	348s	325s	293s

dsp=200 partições, spp=5, ..., 100 partições

está associada ao número de nós utilizados nas execuções e ao domínio das regras de transformação, os quais são descritos a seguir:

- a regra DBLP2ICMT1 encontra todos os autores que publicaram na Conferência Internacional de Transformação de Modelos (ICMT) e a quantidade das respectivas publicações;
- a regra DBLP2ICMT2 identifica se os autores do ICMT ainda estão publicando (ativo) ou se estão inativos (inativo significa que esses autores não publicaram nos últimos 5 anos);
- Já a regra DBLP2ICMT3 busca as conferências, nas quais as pessoas que deixaram de publicar no ICMT voltaram a publicar.

Tabela 5.10: Transformações do Modelo DBLP

Dataset	Tamanho	Elmtos	Nós	SP	DBLP2ICMT1	DBLP2ICMT2	DBLP2ICMT3
DBLP	1.2 GB	5654916	#1	dsp	731s	559s	604s
				spp	651s	497s	539s
			#2	dsp	556s	434s	469s
				spp	498s	388s	419s
			#4	dsp	457s	356s	383s
				spp	410s	315s	339s
			#6	dsp	390s	301s	324s
				spp	367s	286s	305s
			#8	dsp	379s	295s	315s
				spp	353s	279s	288s

dsp=200 partições e spp=600 partições

Nesta seção, foram apresentados os resultados das execuções paralela em modo local, envolvendo a fragmentação, extração e transformação de quatro tipos diferentes de modelos. Os resultados mostram que a Dc4MT é factível em relação a transformação paralela de modelos e escalável em ambientes com paralelismo implícito. A capacidade de fragmentação de modelos e o processamento desses fragmentos em modo paralelo favorecem a escalabilidade presente na Dc4MT, mas a escalabilidade apresentada é dependente de

recursos do ambiente local. Na próxima seção, há uma validação em relação aos resultados dos experimentos em modo local com uma outra abordagem semelhante.

5.2.1 Considerações sobre os resultados - Execuções paralela em modo local

As considerações sobre os resultados apresentados nesta seção têm como ponto de partida, a abordagem Lintra. Essa abordagem foi proposta por Burgueño (2016) e executa transformações de modelos em um ambiente paralelo. Os modelos Imdb e DBLP e o domínio de transformação sob esses modelos são utilizados em ambas abordagens (Lintra e Dc4MT). Os resultados dos experimentos das abordagens Lintra e Dc4MT mostram que o paralelismo favorece o desempenho das execuções, mesmo considerando as limitações do ambiente, como o compartilhamento de memória. Essas abordagens permitem escalabilidade de acordo com a disponibilidade de recursos do ambiente. Quanto ao desempenho das transformações, a abordagem Lintra mensura somente os tempos de execuções das transformações. Ao passo que nos experimentos com a Dc4MT, são medidos os tempos das operações de Fragmentação, Extração e Transformação. Em Lintra, a transformação de modelos é dependente dos modelos em memória e a extração/carga de modelos não é considerada. Sob os aspectos de paralelismo implícito, escalabilidade e transformação paralela de modelos, os resultados obtidos na Dc4MT ratificam a abordagem Lintra. No entanto, a Lintra apresenta diferentes tipos de transformações como M2T, T2M e M2M, enquanto que as transformações de modelos experimentadas na Dc4MT são do tipo M2M.

Os aspectos comuns entre a Lintra e Dc4MT são um ponto positivo, mas não o suficiente. Ao contrário da Lintra, a Dc4MT provê uma estratégia de fragmentação de modelos de modo que os fragmentos são processados em paralelo como *streaming* sem a necessidade de inserir o modelo de entrada em memória de uma só vez. A Dc4MT suporta modelos em formatos XMI e JSON, a Lintra somente XMI. A extração do modelo de entrada para um domínio de modelagem distinto, neste caso em grafo, propicia uma visualização do modelo de entrada sob o aspecto de complexidade estrutural. O modelo de entrada em formato grafo possibilita ter uma dimensão quantitativa de modelo por meio da contabilização de vértices e arestas, corroborando com a visualização da complexidade estrutural do modelo. Além disso, outras análises do modelo de entrada podem ser realizadas com a utilização de algoritmos projetados para processar grafos. Na Dc4MT, há a opção de utilização de uma simples parametrização relacionada ao tamanho do modelo de entrada a ser processado. Trata-se de um parâmetro que interfere no particionamento de dados da operação `shuffle` do ambiente de execução, aumentando o desempenho do processamento paralelo das transformações de modelos. A Dc4MT proporciona a escolha de operações sobre o modelo de entrada. Por exemplo, executar apenas a fragmentação de modelos, de modo que os fragmentos possam ser utilizados em diferentes propósitos. Na próxima seção, os resultados com a transformação distribuída de modelos são apresentados.

5.3 TRANSFORMAÇÕES DISTRIBUÍDAS DE MODELOS

Nesta seção, são apresentados os resultados das execuções em um ambiente distribuído, essas execuções buscam a factibilidade da Dc4MT em relação a escalabilidade nas transformações de VLMs. Os resultados apresentados são decorrentes de 235 execuções submetidas a um *cluster* instanciado na Google Cloud Platform[18] (GCP) formado por um

[18] https://cloud.google.com/

conjunto de 7 máquinas. Desse conjunto, uma máquina tem o papel de nó Master, e as demais de nó Worker (executores). Esse *cluster* é homogêneo e de propósito geral, cujas máquinas são padronizadas (sem cpu de alto desempenho) contendo 4 vCPUs com 15 GB de memória. Um *bucket* (sem SSD *Solid-State Drive*) de armazenamento de 500 GB é associado a uma das máquinas Worker para o armazenamento dos modelos de entrada e do código fonte da transformação (JAR), de modo que essa máquina é usada como um *namespace* do *cluster*. A imagem utilizada nas instâncias é composta pelo sistema operacional Debian 9.0 e os frameworks Hadoop 2.9 com YARN[19] e Spark 2.4.

As submissões das execuções (*Jobs*) foram efetuadas por meio de um console da GCP. A configuração de cada execução é atribuída por um conjunto de argumentos, o qual é enviado via console e interpretado pelo programa `Main.scala`, implementado para as experimentações da Dc4MT. Assim como nas execuções em modo local, o tempo de cada execução é mensurado em segundos com o auxílio da função `System.currentTimeMillis(/1000.000)`.

Os modelos e os domínios de transformações utilizados nas execuções distribuídas são aqueles descritos e utilizados nos experimentos em modo paralelo (seção anterior). Nas execuções distribuídas, foram considerados quatro composições diferentes para o *cluster*, composições com dois, quatro, cinco e seis Workers (#2, #4, #5, #6). Essas composições foram utilizadas nas execuções de cada modelo, com a exceção para os modelos em que se utilizou o *Autoscaling*. O *Autoscaling*[20] é um serviço de provisionamento dinâmico que atua durante o processamento, de modo que os recursos (e.g., executores, memória, controle de *throughput*, *overhead*, ...) são aplicados de forma balanceada a fim de garantir desempenho e evitar falhas de execução. Os modelos que não foram executados em *Autoscaling*, foram submetidos à duas execuções em cada composição do *cluster*, uma com a parametrização padrão do *default shuffle partition* (`dsp=200`) e outra fornecendo um parâmetro via argumento ao *Shuffle Partition Parameter* (`ssp`). Os valores dos parâmetros `ssp` utilizados nas execuções em paralelo foram mantidos nas execuções distribuídas, com uma exceção para os valores iguais à 5, os quais foram substituídos por 7, uma vez que o ambiente distribuído possuí 6 nós Workers ($N_{no} + 1$). As execuções com diferentes composições de nós Workers visam demonstrar a escalabilidade da abordagem Dc4MT.

Na Figura 5.4, é apresentado o resultado das execuções distribuídas das transformações dos modelos Families (`Families2Persons`). Para a execução do modelo Family-11^7-4 foi adotado o *Autoscaling*, uma vez que a configuração padronizada dos recursos do *cluster* poderiam não ser suficientes para o processamento da transformação desse modelo devido ao seu tamanho (19 GB). Na Tabela 5.11, encontram-se os resultados das execuções, os quais são apresentados em modo grafo na Figura 5.4. Observa-se que há um único valor de resultado da execução do modelo Family-11^7-4, isso é em decorrência do Autoscaling adotado na execução, em que a composição de nós é abstraída pelo provisionamento dinâmico de recursos da GCP.

---
[19]http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html
[20]https://cloud.google.com/dataproc/docs/concepts/ configuring-clusters/autoscaling#autoscaling/

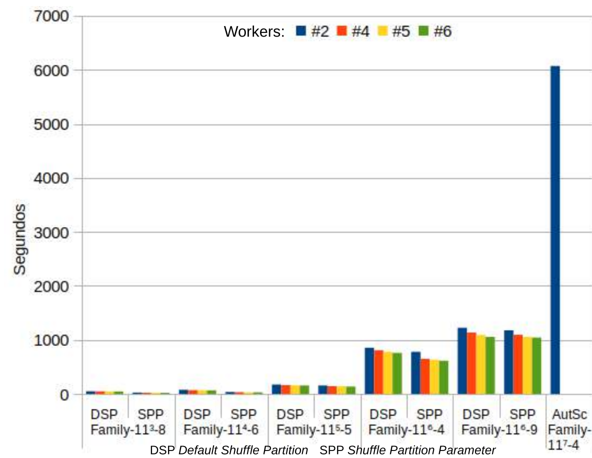


Figura 5.4: Resultado das Transformações Distribuídas de `Families2Persons`

Tabela 5.11: Transformações Distribuídas Families2Persons

Dataset	SP	Quantidade de Nós Workers			
		#2	#4	#5	#6
Family-11^3-8	dsp	49,70s	47,31s	46,64s	45,38s
	spp=7	24,02s	22,04s	21,44s	19,65s
Family-11^4-6	dsp	75,76s	69,70s	67,50s	65,25s
	spp=7	35,15s	31,87s	29,45s	28,06s
Family-11^5-5	dsp	173,80s	163,02s	157,32s	155,87s
	spp=100	155,59s	142,02s	139,03s	135,40s
Family-11^6-4	dsp	856,28s	809,13s	777,95s	760,46s
	spp=600	780,07s	649,42s	632,35s	613,33s
Family-10^6-9	dsp	1224,48s	1136,85s	1091,38s	1056,10s
	spp=700	1177,95s	1095,33s	1055,90s	1043,55s
Family-11^7-4	*Autoscaling*	6071,48s			

`dsp=200` partições `ssp=7, ..., 700` partições

Na Figura 5.5, são exibidos os resultados das transformações distribuídas `Class2Relational`. O *Autoscaling* é adotado na execução da transformação do modelo Clas-11^5-6 em virtude do tamanho, complexidade estrutural desse modelo, e também do domínio de transformação (um conjunto de filtros e sete regras). Os valores em segundos que compõem os resultados dessas transformações estão na Tabela 5.12.

Tabela 5.12: Transformações Distribuídas Class2Relational

Datasets	SP	Quantidade de Nós Workers			
		#2	#4	#5	#6
Clas-11^5-0	dsp	80,58s	63,28s	56,51s	54,09s
	spp=7	57,89s	41,62s	38,98s	36,65s
Clas-11^5-1	dsp	1421,06s	946,06s	891,77s	837,48s
	spp=100	1299,04s	857,49s	809,60s	761,73s
Clas-11^5-2	dsp	1913,94s	1468,26s	1337,39s	1308,76s
	spp=150	1631,86s	1345,57s	1214,69s	1181,97s
Clas-11^5-3	dsp	2630,81s	2189,36s	2086,81s	1984,25s
	spp=300	2266,65s	1875,70s	1759,25s	1738,46s
Clas-11^5-4	dsp	3003,49s	2580,68s	2385,24s	2341,36s
	spp=400	2825,14s	2383,08s	2194,21s	2162,06s
Clas-11^5-5	dsp	4150,27s	3528,37s	3381,54s	3234,70s
	spp=500	3981,04s	3382,85s	3118,82s	3052,83s
Clas-11^5-6	*Autoscaling*	5271,18s			

`dsp=200` partições `ssp=7, ..., 500` partições

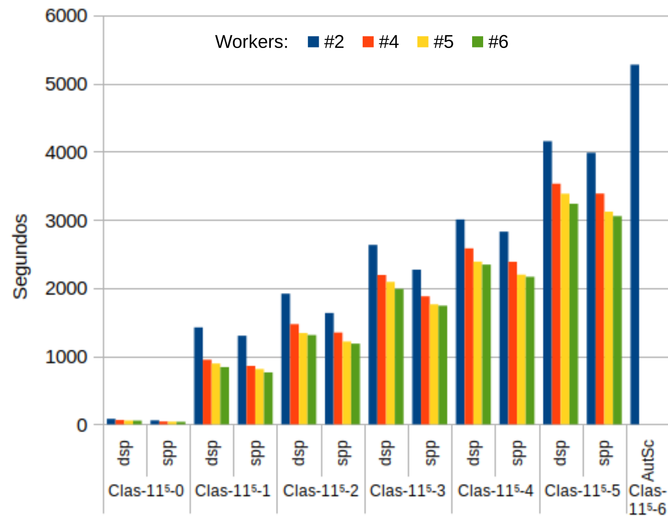
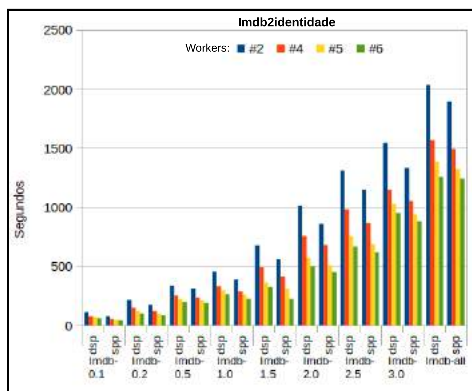
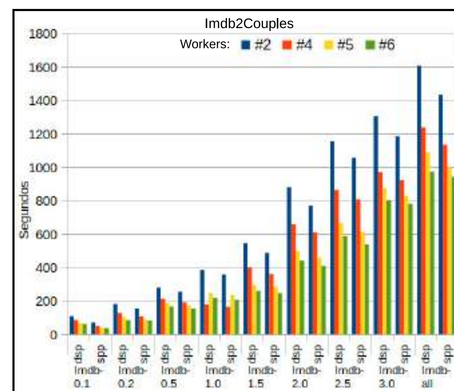


Figura 5.5: Resultado das Transformações Distribuídas de `Class2Relational`

Na Figura 5.6, é ilustrado o resultado das transformações distribuídas dos modelos Imdb. Nas Sub figuras 5.6(a) e 5.6(b) são apresentados os resultados das transformações `Imdb2Identidade` e `Imdb2Couples`. O domínio dessas transformações foram apresentados na Seção 5.2, assim como o domínio das transformações do modelo DBLP. Na Tabela 5.13, são apresentados os valores em segundos das transformações distribuídas dos modelos Imdb. O *Autoscaling* também foi adotado nas execuções das transformações do modelo DBLP em razão da complexidade estrutural e do tamanho desse modelo, cujos resultados das execuções estão na Tabela 5.14.



(a) Transformações Distribuídas `Imdb2Identidade`



(b) Transformações Distribuídas `Imdb2Couples`

Figura 5.6: Transformações Distribuídas dos Modelos `Imdb`

Os resultados dos experimentos contidos nas Tabelas 5.11, 5.12 e 5.13 são observados sob duas perspectivas: escalabilidade e a parametrização do `shuffle.partition`. A escalabilidade é analisada considerando a média dos tempos das execuções obtida em dois nós Workers (coluna #2) comparada com a média dos tempos das execuções em quatro nós Workers (coluna #4), e por sua vez é comparada com a média dos tempos das execuções em seis nós Workers (coluna #6). Assim, o decréscimo médio de tempos das execuções em dois nós Workers em relação à quatro nós Workers é de 21% e entre quatro nós Workers e seis nós Workers é de 34%. Isto significa que a medida em que nós Workers (máquinas) são inseridos ao ambiente distribuído o desempenho aumenta.

Tabela 5.13: Transformação Distribuída Imdb

Datasets	SP	Imdb2identidade – #Workers				FindCouple – #Workers			
		#2	#4	#5	#6	#2	#4	#5	#6
Imdb-0.1	dsp	115,63s	79,84s	72,95s	64,7s	110,08s	86,11s	67,54s	62,19s
	spp=7	80,72s	56,98s	46,3s	42,73s	71,99s	49,63s	39,41s	36,95s
Imdb-0.2	dsp	217,43s	151,82s	123,71s	101,21s	182,57s	127,80s	102,91s	84,65s
	spp=7	176,95s	123,67s	98,94s	91,00s	156,62s	109,10s	89,75s	81,95s
Imdb-0.5	dsp	337,01s	253,31s	226,88s	200,44s	281,05s	212,84s	189,32s	167,85s
	spp=7	313,17s	234,87s	211,85s	191,12s	255,97s	191,97s	173,98s	155,83s
Imdb-1.0	dsp	458,98s	333,08s	296,37s	264,89s	385,08s	179,01s	249,37s	220,36s
	spp=10	393,54s	290,11s	262,36s	227,04s	359,96s	165,74s	237,17s	207,76s
Imdb-1.5	dsp	676,33s	494,46s	366,58s	323,95s	545,98s	398,27s	294,74s	260,92s
	spp=30	560,04s	417,39s	311,72s	227,38s	486,73s	363,23s	287,23s	248,15s
Imdb-2.0	dsp	1014,38s	759,32s	574,62s	504,25s	881,37s	659,02s	498,28s	439,34s
	spp=40	860,49s	680,2s	510,83s	453,46s	771,13s	611,20s	459,05s	407,47s
Imdb-2.5	dsp	1309,66s	980,73s	754,41s	666,9s	1153,26s	864,25s	664,17s	586,35s
	spp=60	1147,52s	866,44s	689,67s	620,12s	1058,01s	807,49s	613,18s	536,96s
Imdb-3.0	dsp	1543,75s	1146,88s	1031,25s	950,04s	1305,58s	970,81s	873,17s	803,43s
	spp=80	1335,64s	1054,76s	938,23s	881,46s	1182,83s	921,76s	828,54s	781,11s
Imdb-all	dsp	2036,34s	1567,49s	1385,55s	1255,1s	1608,22s	1237,32s	1092,43s	973,62s
	spp=100	1893,00s	1493,53s	1327,98s	1241,61s	1434,36s	1133,01s	998,40s	943,46s

dsp=200 partições ssp=7, ..., 100 partições

Tabela 5.14: Transformações Distribuídas DBLP

Dataset	SP	DBLP2ICMT1	DBLP2ICMT2	DBLP2ICMT3
DBLP	Autoscaling	2483,86s	2049,37s	1944,07s

A inserção de nós se mostrou mais significativa ao desempenho, quando essa inserção é feita em pares (dois nós por vez), devido as características do ambiente distribuído (e.g., latência, *overhead*, concorrência, entre outras). O uso de parâmetros `ssp` é observado em relação a parametrização padrão (`dsp`) que é de 200 partições. Desta forma, a média de tempos é computada para os modelos executados com parâmetros menores que 200 e para modelos executados com parâmetros maiores que 200 ($spp < 200$ $ssp > 200$). Os resultados estão na Tabela 5.15 sem os resultados das execuções em que o *Autoscaling* foi adotado, já que nenhum tipo de parametrização foi adicionado para essas execuções.

Tabela 5.15: Impacto do *Shuffle Partition* nas Execuções Distribuídas das Transformações

spp/dsp	Families	Class	Imdb
spp < 200	41%	17%	13,5%
spp > 200	10%	9%	0%
Média-Modelos	25,5%	13%	13,5%

De acordo com as porcentagens exibidas na Tabela 5.15, o uso do `ssp` é mais efetivo quando há a seguinte combinação: modelos menores, com complexidade estrutural baixa e domínio de transformação simplificado (e.g., parte dos modelos Families). De qualquer modo, o uso do `ssp` evidencia um melhor desempenho também nas execuções distribuídas, minimizando os limitadores de eficiência presentes nos sistemas distribuídos como latência, *throughput*, tempo de resposta, entre outros. Os resultados obtidos com os experimentos em um ambiente distribuído mostram que a Dc4MT é escalável, mas ao mesmo tempo revelam que provisionamento do *cluster* não foi adequado, comprometendo o desempenho das transformações quando comparados com os resultados das execuções em paralelo local.

5.3.1 Considerações sobre os resultados - Execuções Distribuídas

ATL-MR é uma abordagem proposta por Benelallam (2016), implementada em ATL sob o modelo MapReduce e executada em um *cluster* com 12 máquinas, utilizando o framework Hadoop com YARN. Entre os diferentes tipos de transformações experimentadas nessa abordagem, há transformações *Class2Relational*, cujos modelos Class possuem entre 5000 a 20000 elementos com uma densidade de referências igual 8. Os modelos do tipo Class utilizados nos experimentos da Dc4MT são maiores e possuem uma densidade de referencias entre 0 à 6. Não é intuito nesta seção apresentar uma comparação dos resultados obtidos nos experimentos de ambas abordagens em relação a transformação *Class2Relational*, mas destacar aspectos comuns, tais como: escalabilidade, particionamento, parametrização e customização do ambiente distribuído e o impacto de operações não monotônicas[21]

De acordo com os resultados dos experimentos, ambas abordagens são escaláveis. Quanto ao particionamento, a ATL-MR utiliza um algoritmo baseado em grafo de dependência para realizar o particionamento de modelos de entrada. O resultado do particionamento está sujeito a falsos positivos, em que elementos do modelo podem ser direcionados à partições indevidas, desbalanceando o particionamento e comprometendo o desempenho da transformação. O desempenho pode ser ainda degradado em relação aos elementos do modelo com grande quantidade de dependência ou ainda elementos que são transmitidos pelo nó Master aos nós Workers em ordem desfavorável ao processamento da transformação. Isso significa que o algoritmo de particionamento é executado no nó Master, e enquanto a distribuição das partições não é finalizada, a transformação não é concluída. Na Dc4MT, a operação de fragmentação do modelo de entrada também é executada no nó Master, gerando blocos de fragmentos que facilitam o processamento distribuído desses blocos independente da ordem que eles são acessados pelos nós Workers. Internamente esses blocos são particionados pelo modelo de comunicação do paralelismo implícito. A Dc4MT utiliza um parâmetro, em que o valor desse parâmetro é definido de acordo com o tamanho de cada modelo de entrada como uma forma de interferir no particionamento durante processamento desses modelos. Esse parâmetro altera a quantidade de partições que são transmitidas durante a troca de mensagens entre os nós Workers durante a execução da operação *shuffle*, minimizando a dependência de dados (neste caso do tipo *Wide*). Em ambas estratégias de particionamento utilizadas pelas as abordagens ATL-MR e Dc4MT, há ganhos de desempenho no processamento das transformações distribuídas.

Nas discussões dos experimentos em ATL-MR, é argumentado sobre a necessidade de parametrização e customização do ambiente distribuído. Essa necessidade também foi identificada durante os experimentos com a Dc4MT no ambiente distribuído, em que foi necessário o uso do *Autoscaling* como uma alternativa à parametrização. Os aspectos como tamanho e complexidade estrutural do modelo e o domínio de transformação presentes na Dc4MT, devem ser considerados no processo de parametrização e customização do ambiente distribuído, além das características do próprio ambiente. As operações do tipo não-monotônicas (e.g., agregação) estão presentes em ambas abordagens: `collect()` em ATL-MR e `coalesce()` em Dc4MT. Essas operações podem fazer parte de filtros e regras de transformação, agregando seus resultados na composição do modelo de saída. Esse tipo de operação exige a sincronização de tarefas em execução nos nós Workers, potencializando os limitadores de desempenho em aplicações distribuídas em ambas abordagens.

---

[21]não monotonicidade é a propriedade que ao adicionar um elemento a um conjunto de entrada pode revogar um elemento anteriormente válido de um conjunto de saída. É necessário a coordenação entre os elementos, garantindo que as entradas sejam concluídas (Alvaro et al., 2011).

5.4 CONSIDERAÇÕES FINAIS

Neste capítulo, foram apresentados os resultados dos experimentos, em que uma implementação da abordagem Dc4MT é validada sob as perspectivas factibilidade, escalabilidade e desempenho. Os experimentos foram executados em ambientes de processamento paralelo local e distribuído. De acordo com os resultados, a Dc4MT é escalável e processa transformações de VLMs em ambientes paralelo e distribuído. Um conjunto de modelos com diferentes tamanhos e complexidade estrutural distintas foram utilizados como entrada nos experimentos, e de acordo com a característica de cada modelo, um diversificado conjunto de regras de transformações foi especificado e executado. As operações de fragmentação e de extração foram experimentadas em execuções separadas em modo local, uma vez que a fragmentação é executada no nó Master como um *sequential parsing* sobre o modelo de entrada gerando fragmentos desse modelo. Os experimentos com a operação de extração mostraram o aspecto *lazy-evaluate* presente no ambiente de execução, como também o tipo de grafo gerado por meio da contabilização da quantidade de vértices e arestas presentes nos grafos gerados. Além disso, a operação de extração foi utilizada no comparativo entre os modelos Families expressados nos formatos XMI e JSON. As versões 1.0 e 2.0 do formato XMI são suportadas pela operação de extração.

Os resultados obtidos com os experimentos revelaram que a escalabilidade no processamento das transformações de modelos é mais presente no ambiente distribuído. No entanto, o desempenho das transformações distribuídas quando comparado com o desempenho das transformações em paralelo é bastante inferior. Isto é decorrente do sub provisionamento do *cluster* utilizado nos experimentos com as execuções distribuídas. Nesse provisionamento foi considerado somente o tamanho do modelo (complexidade estrutural do modelo de entrada e o domínio da transformação foram ignorados) em relação ao total de memória dos nós Workers. As questões que limitam o desempenho em sistemas distribuídos não foram analisadas. Sobretudo, em ambientes de paralelismo implícito que recorrem à camadas de software (e.g., Hadoop, Yarn, Spark) para tornar factível tal paralelismo. Nesses ambientes, há uma demanda pré requisitada de recursos que tem que ser adicionada ao provisionamento, além do que se pretende processar. O paralelismo implícito é uma abordagem bem difundida no processamento intensivo de dados e pode ser considerada viável ao processamento de VLMs, mas é necessário considerar os custos envolvidos, principalmente em ambientes distribuídos. Os resultados também revelaram que a intervenção no particionamento de dados na operação *shuffle partition* do ambiente de execução, por meio do parâmetro (`ssp`) durante processamento paralelo ou distribuído das transformações de modelos, propicia um aumento significativo no desempenho das transformações em relação ao desempenho das transformações com parâmetro padrão do ambiente (`dsp`). No entanto, fica em aberto o quanto essa intervenção pode favorecer o desempenho.

6 CONCLUSÃO

Operações sobre modelos são uma tarefa árdua, dado os aspectos como complexidade estrutural do modelo e a quantidade de elementos a ser processado. A transformação de modelos é uma dessas operações, que além das características do modelo, lida com o domínio da transformação em questão. Nesta tese, foi proposto uma abordagem para a transformação paralela e distribuída de modelos baseada nas abordagens MDE e Dc. A MDE é uma abordagem da ES utilizada no desenvolvimento de software e focada em modelos, e a Dc é uma abordagem orientada a dados, em que o dado é o elemento principal. A Dc4MT foi proposta unindo aspectos dessas abordagens heterogêneas, tendo como propósito principal a transformação escalável de VLMs. Esse propósito está alinhado com o objetivo principal desta tese que foi propor uma abordagem orientada a dados para a transformação paralela e distribuída de modelos, visando a escalabilidade.

A concretização da Dc4MT levou em conta a questão principal de pesquisa, da qual a abordagem proposta foi implementada em uma plataforma escalável, provendo operações de fragmentação, extração e transformação de VLMs em uma única plataforma. Desse modo, as questões complementares RQ1, RQ2 e RQ3 são respondidas, assim como os objetivos secundários são alcançados com as seguintes contribuições:

- a fragmentação de modelos é baseada em uma estratégia que gera fragmentos balanceados de modelos, qualificando esses fragmentos ao processamento paralelo ou distribuído sem a necessidade de ordenação ou rearranjo para tal processamento. A estratégia adotada permite percorrer o modelo a ser fragmentado sem a obrigação de retê-lo por completo em memória. O resultado da fragmentação de modelos é fundamental para a operação de extração da Dc4MT, cujos fragmentos são a entrada para essa operação;

- a extração de modelos proposta para a Dc4MT foi elaborada de maneira que os elementos do modelo de entrada são extraídos dos fragmentos com o processamento paralelo ou distribuído desses fragmentos sem a presença de todos os fragmentos em memória principal. Para tanto, os fragmentos são atribuídos aos executores (nós) do ambiente paralelo/distribuído e processados. O resultado da extração é um grafo acíclico formado por um conjunto de vértices e arestas representado em um GraphFrames. A representação em grafo é a evidência da tradução do domínio de modelagem do modelo de entrada expressado em XMI (versões 1,0 e 2,0) ou JSON para um novo domínio de modelagem. Ter o modelo de entrada representado em grafo foi uma escolha pautada nas seguintes premissas: modelos em geral possuem uma estrutura semelhante a um grafo; o modelo de entrada traduzido para um grafo permite operações além da transformação, como a visualização da densidade de referências entre os elementos do modelo, a quantificação do modelo em termos de vértices e arestas, a identificação do tipo de grafo, entre outras operações específicas sobre grafos que podem ser exploradas;

- resolução de referências: a extração de modelos trata eficientemente os elementos que possuem referência(s) hierárquica(s) *containment*, distribuindo essas referências em arestas. No entanto, há modelos com elementos que possuem uma ou mais referências a outro(s) elemento(s) independente de sua posição hierárquica. Essas

referências são preservadas pela extração de modelos, mas não são resolvidas. Assim, um algoritmo de resolução de referências foi inserido no módulo Extrator, de modo que novos vértices e arestas são adicionados ao grafo, estratificando os vértices que contém strings com referências entre os elementos do modelo de entrada;

- a transformação de modelos foi especificada em estilo declarativo contendo filtros e regras, os quais são submetidos sobre os vértices e arestas para produzir a transformação (as regras de transformação estão no Apêndice A). No entanto, a maioria das instruções que formam os filtros e as regras de transformação utiliza junções entre os vértices e arestas (`join()`), uma vez que os elementos e seus atributos estão estruturados em um grafo. O uso de junção tem impacto direto na troca de mensagens entre os nós de um ambiente paralelo ou distribuído, tornando o processamento da transformação de modelos custoso. Para mitigar esse efeito, um parâmetro foi usado como um método de interferência ao particionamento de dados realizado pelo ambiente de paralelismo implícito adotado na implementação da Dc4MT. Os resultados com os experimentos, usando esse método, mostraram um aumento médio de 19% no desempenho das execuções (paralela local) das transformações de modelos em relação as mesmas execuções sem a interferência no particionamento (dsp). Neste sentido, o desempenho médio das execuções distribuídas das transformações de modelos foi de 17,35% .

As contribuições em destaque são o resultado de uma implementação da Dc4MT unificada em uma plataforma escalável. A Dc4MT é passível de implementações em outras plataformas de paralelismo implícito, mas nesta tese o framework Spark e suas APIs RDD, DataFrame e GraphFrames foram escolhidos porque contemplam os requisitos da Dc4MT, unindo duas abordagens distintas, a MDE e a Dc.

A junção dessas abordagens no âmbito de transformação de modelos foi reportada na 34*th* ACM/SIGAPP *Symposium on Applied Computing* (SAC) de 2019 por meio do artigo **Applying a Data-centric Framework for Developing Model Transformations** dos autores Camargo e Fabro (2019).

Um outro artigo, intitulado **A Data-centric Model Transformation Approach using Model2GraphFrames Transformations** foi submetido por esses autores ao JSERD (*Journal of Software Engineering and Research Development*) em Setembro de 2019. Os resultados apresentados pela abordagem Dc4MT são promissores. No entanto, ainda há espaços para incorporar melhorias à Dc4MT, os quais são apresentados na próxima seção.

6.1 TRABALHOS FUTUROS

Trabalhos futuros no âmbito desta tese incluem: inserir diretivas ao módulo Fragmentador, de modo que se possa fragmentar modelos em formato texto, como pseudo código e código fonte, entre outros; estender essas diretivas ao Módulo Extrator, amplificando o domínio de tradução de modelos para o formato grafo, além do XMI e JSON. Além disso, é possível viabilizar a fragmentação de modelos para outros domínios de aplicações como a construção e manipulação colaborativa de VLMs, visto que nesta tese a fragmentação foi direcionada a extração e a transformação de modelos; realizar novos experimentos com transformações M2T e T2M; ainda no módulo Extrator, inserir diretivas para tratar as referências entre elementos de modelos baseadas em UUID (*Universally Unique Identifier*);

propor uma estratégia para o particionamento em memória de GraphFrames, de maneira que se possa executar junções (`join()`) em modo local (nó Worker), minimizando a dependência de dados e consequentemente a troca de mensagens entre nós durante a execução das transformações. Algoritmos de clusterização como os propostos por (Fan et al., 2020; Zeng e Yu, 2018; Guha et al., 2017) podem ser alternativas para compor uma estratégia de particionamento de grafos em memória. Ademais, outras estratégias de particionamento podem ser inseridas buscando uma customização do particionamento, além do *Shuffle Partition*, visto que nos experimentos com modelos grandes a parametrização do *Shuffle Partition* indicou uma possível estagnação em relação à melhoria de desempenho que precisa ser melhor investigada.

Inserir a visualização do modelo de entrada em modo grafo como uma percepção da complexidade estrutural do modelo a ser processado. Essa percepção, somada ao tamanho do modelo e ao domínio da transformação podem servir como indicadores no provisionamento de ambientes paralelo e distribuído. Esses indicadores podem ser incorporados como um novo módulo da Dc4MT, como também em outras abordagens de processamento paralelo/distribuído de modelos. Por exemplo, nos experimentos realizados com execuções de transformações distribuídas, apenas um desses elementos foi considerado, resultando em um sub provisionamento do ambiente distribuído. Em decorrência disso, execuções falharam exigindo novas submissões de execuções e o desempenho das transformações em ambientes distribuídos não foi satisfatório quando comparados aos resultados do ambiente paralelo local. Além disso, ambientes distribuídos exigem configurações adequadas ao que se pretende executar, principalmente aqueles com paralelismo implícito. Tais configurações, devem levar em conta os limitadores de eficiência em sistemas distribuídos (e.g., latência, largura de banda, tempo de resposta e taxa de transferência) e mitigá-los com estratégias direcionadas a melhorar balanceamento de carga, minimizar o *overhead* e o *data skew*.

A ausência de um modelo escalável para a persistência específica de modelos também contribuiu para o baixo desempenho das transformações distribuídas. Um novo módulo pode ser inserido à Dc4MT como uma camada de persistência e acesso escalável de VLMs, incluindo os fragmentos do modelo de entrada e o modelo de saída.

REFERÊNCIAS

Aaronson, S. (2005). Guest column: Np-complete problems and physical reality. *SIGACT News*, 36(1):30–52.

Aderholdt, F., Venkata, M. G. e Parchman, Z. (2018). Sharp data constructs: Data constructs to enable data-centric computing. Em *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, páginas 170–177.

Ahlgren, B., Hidell, M. e Ngai, E. C. (2016). Internet of things for smart cities: Interoperability and open data. *IEEE Internet Computing*, 20(6):52–56.

Aki-Hiro, S. (2014). *Applied Data-Centric Social Sciences: Concepts, Data, Computation, and Theory*, volume 1. Springer Japan, 1 ed. edition.

Almendros-Jiménez, J. M. e Iribarne, L. (2013). *A Model Transformation Language Based on Logic Programming*, páginas 382–394. Springer Berlin Heidelberg, Berlin, Heidelberg.

Almendros-Jiménez, J. M., Iribarne, L., López-Fernández, J. e Mora-Segura, A. (2016). Ptl: A model transformation language based on logic programming. *Journal of Logical and Algebraic Methods in Programming*, 85(2):332 – 366.

Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., ellerstein, J. M. e Sears, R. (2010). Boom analytics: Exploring data-centric, declarative programming for the cloud. Em *Proceedings of the 5th European Conference on Computer Systems*, páginas 223–236, New York, NY, USA. ACM.

Alvaro, P., Conway, N., Hellerstein, J. M. e Marczak, W. R. (2011). Consistency analysis in bloom: a CALM and collected approach. Em *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, páginas 249–260.

Amálio, N., de Lara, J. e Guerra, E. (2015). Fragmenta: A theory of fragmentation for mde. Em *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, páginas 106–115.

Apache, A. S. F. (2019a). Storm apache. `https://storm.apache.org/`. Accessado em 10/11/2019.

Apache, A. S. F. (2019b). Velocity apache 2017/ v 2.1. `http://velocity.apache.org//`. Acessado em 11/2019.

Apache, S. F. (2019c). Apache, hadoop release 3.1.2. `http://hadoop.apache.org/`. Online, accessado 09-2019.

Apache, S. F. (2019d). Apache spark, release 2.4.3. `https://spark.apache.org/`. Online, accessed 2019-9.

Apache, S. F. (2019e). Gremlin, tinkerpop release 3.4.4. `https://tinkerpop.apache.org/gremlin.html/`. Online, acessado 10/2019.

Aracil, J. M. P. e Ruiz, D. S. (2016). Towards distributed ecore models. Em *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, páginas 209–216.

Aracil, P. M. J. e Ruiz, S. D. (2017). Cloudtl: A new transformation language based on big data tools and the cloud. Em *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*, MODELSWARD 2017, páginas 137–146, Portugal. SCITEPRESS - Science and Technology Publications, Lda.

Babau, J.-P., Mireille, B.-F., Champeau, J., Sylvain, R. e A., S. (2010). *Model-Driven Engineering for Distributed Real-Time Systems: MARTE Modeling, Model Transformations and their Usages*, volume 1. ISTE Ltd and John Wiley & Sons, Inc, 1 ed. edition.

Basso, F. P., Oliveira, T. C., Werner, C. M. L. e Becker, L. B. (2017). Building the foundations for 'mde as service'. *IET Software*, 11(4):195–206.

Batory, D. e Azanza, M. (2017). *Teaching model-driven engineering from a relational database perspective*, volume 16, páginas 443–467. Springer Berlin Heidelberg, Berlin, Heidelberg.

Batory, D., Latimer, E. e Azanza, M. (2013). Teaching model driven engineering from a relational database perspective. Em Moreira, A., Schätz, B., Gray, J., Vallecillo, A. e Clarke, P., editores, *Model-Driven Engineering Languages and Systems*, páginas 121–137, Berlin, Heidelberg. Springer Berlin Heidelberg.

Benelallam, A. (2016). *Model transformation on distributed platforms: decentralized persistence and distributed processing*. Tese de doutorado, Universite Bretagne Loire, Laboratoire d' informatique de Nantes-Atlantique (LINA).

Benelallam, A., Gómez, A., Tisi, M. e Cabot, J. (2015). Distributed model-to-model transformation with atl on mapreduce. Em *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, páginas 37–48, New York, NY, USA. ACM.

Benelallam, A., Gómez, A., Tisi, M. e Cabot, J. (2018). Distributing relational model transformation on mapreduce. *Journal of Systems and Software*, 142:1 – 20.

Benelallam, A., Tisi, M., Cuadrado, J. S., de Lara, J. e Cabot, J. (2016). Efficient model partitioning for distributed model transformations. *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering - SLE 2016*, páginas 226–238.

Benelallam, A., Tisi, M., Rath, I., Izso, B. e Kolovos, D. (2014). Towards an open set of real-world benchmarks for model queries and transformations. Em *CEUR Workshop Proceedings*, volume 1206, página 8.

Berramla, K., Deba, E. A. e Benhamamouch, D. (2016). Model transformation generation a survey of the state-of-the-art. Em *2016 International Conference on Information Technology for Organizations Development (IT4OD)*, páginas 1–6.

Bézivin, J., Jouault, F., Rosenthal, P. e Valduriez, P. (2005). Modeling in the large and modeling in the small. Em *Proceedings of the 2003 European Conference on Model*

Driven Architecture: Foundations and Applications, MDAFA'03, páginas 33–46, Berlin, Heidelberg.

Blouin, A., Moha, N., Baudry, B., Houari, A. S. e Marc, j. E. (2015). Assessing the use of slicing-based visualizing techniques on the understanding of large metamodels. *Information and Software Technology*, 62:124–142.

Blouin, A., Moha, N., Baudry, B. e Sahraoui, H. (2014). Slicing-based techniques for visualizing large metamodels. Em *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*, páginas 25–29. IEEE Computer Society.

Bonifati, A., Mecca, G., Papotti, P. e Velegrakis, Y. (2011). Discovery and correctness of schema mapping transformations. Em Bellahsene, Z., Bonifati, A. e Rahm, E., editores, *Schema Matching and Mapping*, páginas 111–147. Springer Berlin Heidelberg, Berlin, Heidelberg.

BOOM, R. G. (2013). Bloom, berkeley language orders of magnitude, released 0.9.7. `http://bloom-lang.net//`. Acessado em 14-Oct-2019.

Brambilla, M., Cabot, J. e Wimmer, M. (2017). *Model-Driven Software Engineering in Practice*, volume 1. Morgan & Claypool, 2 ed. edition.

Breitbart, Y., Deacon, A., Schek, H.-J., Sheth, A. e Weikum, G. (1993). Merging application-centric and data-centric approaches to support transaction-oriented multi-system workflows. *ACM Sigmod Record*, 22(3):23–30.

Bucchiarone, A., Cabot, J., Paige, R. F. e Pierantonio, A. (2020). Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling*.

Burgueño, L., Troya, J., Wimmer, M. e Vallecillo, A. (2013). On the concurrent execution of model transformations with linda. Em *Proceedings of the Workshop on Scalability in Model Driven Engineering*, BigMDE '13, páginas 3:1–3:10, New York, NY, USA. ACM.

Burgueño, L., Troya, J., Wimmer, M. e Vallecillo, A. (2015). Parallel in-place model transformations with lintra. Em *3rd Workshop on Scalable Model Driven Engineering (BigMDE 2015)*, volume 1406, página 11.

Burgueño, L., Wimmer, M. e Vallecillo, A. (2016). A linda-based platform for the parallel execution of out-place model transformations. *Inf. Software Technology*, 79:17–35.

Burgueno, L. (2013). On the concurrent and distributed model transformations based on linda. Em *DocSymp MoDELS*, páginas 9–16. Citeseer.

Burgueño, L. (2016). *On the Quality Properties of Model Transformations:Performance and Correctness*. Tese de doutorado, Universidad Málaga, Departamento de Lenguajes y Ciencias de la Computacion.

Caíno-Lores, S. e Jesús, C. (2016). A survey on data-centric and data-aware techniques for large scale infrastructures. Em *International Journal of Computer and Information Engineering Vol:10, No:3*.

Calvar, T. L., Jouault, F., Chhel, F. e Clavreul, M. (2019). Efficient atl incremental transformations. *JOT Journal of Object Technology*, 18:1–17.

Camargo, L. C. e Fabro, M. D. D. (2019). Applying a data-centric framework for developing model transformations. Em *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, páginas 1570–1573. ACM.

Chambers, B. e Zaharia, M. (2018). *Spark: The Definitive Guide*, volume 1. Ó Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA, USA, 1 ed. edition.

Cicchetti, A., Di Ruscio, D., Eramo, R. e Pierantonio, A. (2011). Jtl: A bidirectional and change propagating transformation language. Em *Proceedings of the Third International Conference on Software Language Engineering*, páginas 183–202. Springer-Verlag.

Clasen, C., Del Fabro, M. D. e Tisi, M. (2012). Transforming very large models in the cloud: a research roadmap. Em *First International Workshop on Model-Driven Engineering on and for the Cloud*. Springer.

Clements, P. e Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Combemale, B., Kienzle, J. A., Mussbacher, G., Ali, H., Amyot, D., Bagherzadeh, M., Batot, E., Bencomo, N., Benni, B., Bruel, J., Cabot, J., Cheng, B. H. C., Collet, P., Engels, G., Heinrich, R., Jezequel, J., Koziolek, A., Mosser, S., Reussner, R., Sahraoui, H., Saini, R., Sallou, J., Stinckwich, S., Syriani, E. e Wimmer, M. (2020). A hitchhiker's guide to model-driven engineering for data-centric systems. *IEEE Software*, páginas 1–10.

Cuadrado, S. J. e de Lara, J. (2013). Streaming model transformations: Scenarios, challenges and initial solutions. Em Duddy, K. e Kappel, G., editores, *Theory and Practice of Model Transformations: 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, páginas 1–16. Springer Berlin Heidelberg, Berlin, Heidelberg.

da Silva, R. A. (2015). Model-driven engineering. *Comput. Lang. Syst. Struct.*, 43(C):139–155.

Daniel, G. (2017). *Efficient persistence, query, and transformation of large models*. Tese de doutorado, Universite Bretagne Loire, Laboratoire des sciences du numerique de Nantes (LS2N).

Daniel, G., Jouault, F., Sunyé, G. e Cabot, J. (2017). Gremlin-atl: A scalable model transformation framework. Em *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, páginas 462–472, Piscataway, NJ, USA. IEEE Press.

Daniel, Gwendaland Sunyé, G., Benelallam, A., Tisi, M., Vernageau, Y., Gómez, A. e Cabot, J. (2016). Neoemf: a multi-database model persistence framework for very large models. Em *Proceedings of the MoDELS 2016 Demo and Poster Sessions co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), Saint-Malo, France, October 2-7, 2016.*, páginas 1–7.

Dave, A., Jindal, A., Li, L. E., Xin, R., Gonzalez, J. e Zaharia, M. (2016). Graphframes: An integrated api for mixing graph and relational queries. Em *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, páginas 2:1–2:8, New York, NY, USA. ACM.

Davidson, S. G., Boyack, W. K., Zacharski, R. A., Helmreich, S. C. e R., C. J. (2006). Data-centric computing with the netezza architecture. Relatório Técnico SAND2006-3640, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550.

de Murcia, U. (2013). Morsa. `/http://modelum.es/trac/morsa/`. Accessed in 2017/10.

Deák, Mezei, G., Vajk, K. e Fekete, T. (2013). Graph partitioning algorithm for model transformation frameworks. Em *Eurocon 2013*, páginas 475–481.

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

Diaz, D. (1999). Gnu prolog 2017/ v 1.4.4. `http://www.gprolog.org/`. 2017/08.

Dolby, J., Hammer, C., Marino, D., Tip, F., Vaziri, M. e Vitek, J. (2012). A data-centric approach to synchronization. *ACM Trans. Program. Lang. Syst.*, 34(1):4:1–4:48.

Drey, Z., Faucher, C., Fleurey, F., Mahe, V. e Vojtisek, D. (2017). Kermeta language reference manual. `diverse-project.github.io/k3/publish/user_documentation/html_single/user_documentation.html`. Accessed in 2017/07.

Eclipse, F. (2014). Epsilon platform, 2014 / v2.4. `http://www.eclipse.org/epsilon/`. Accessed in 2017/07.

Eclipse, F. (2019a). Atl - transformation language, release 4.1. `https://projects.eclipse.org/projects/modeling.mmt.atl`. online, accessed 2019-09.

Eclipse, F. (2019b). Atl - transformations list. `http://www.eclipse.org/atl/atlTransformations/`. Acessado em = 19/09/2019.

Eclipse, F. (2019c). Emf, eclipse modeling framework, ide2019-9. `http://www.eclipse.org/modeling/emf/`. Accessed in 2019/09.

Fabro, D. D. M. (2007). *Metadata management using model weaving and model transformation.* Tese de doutorado, Universite de Nantes, Laboratoire d'Informatique de Nantes Atlantique, Nantes FRA.

Fan, W., Liu, M., Tian, C., Xu, R. e Jingren, Z. (2020). Incrementalization of graph partitioning algorithms. Em *Proceedings of the VLDB Endowment*, volume 13 de *PVLDB'20*, páginas 1261–1274, Wolfgang Lehner (TU Dresden, Germany).

Fekete, T. e Mezei, G. (2016). Towards a model transformation tool on the top of the opencl framework. Em *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, páginas 355–360.

Ganov, S., Khurshid, S. e Perry, D. E. (2011). A case for alloy annotations for efficient incremental analysis via domain specific solvers. Em *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, páginas 464–467.

García-Magariño, I., Gómez-Sanz, J. J. e Fuentes-Fernández, R. (2009). Model transformation by-example: An algorithm for generating many-to-many transformation rules in several model transformation languages. Em F., P. R., editor, *Theory and Practice of Model Transformations*, páginas 52–66. Springer Berlin Heidelberg.

Garmendia, A., Guerra, E., Kolovos, D. S. e De Lara, J. (2014). EMF splitter: A structured approach to EMF modularity. *CEUR Workshop Proceedings*, 1239:22–31.

Ge, B., Hipel, K. W., Li, L. e Chen, Y. (2012). A data-centric executable modeling approach for system-of-systems architecture. Em *2012 7th International Conference on System of Systems Engineering (SoSE)*, páginas 368–373.

Gelernter, D. (1985). Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112.

Giese, H., Lambers, L., Becker, B., Hildebrandt, S., Neumann, S., Vogel, T. e Wätzoldt, S. (2012). Graph transformations for mde, adaptation, and models at runtime. Em Bernardo, Cortellessa, V. e Pierantonio, A., editores, *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer*, páginas 137–191. Springer Berlin Heidelberg, Berlin, Heidelberg.

Gómez, A., Tisi, M., Sunyé, G. e Cabot, J. (2015). Map-based transparent persistence for very large models. Em Egyed, A. e Schaefer, I., editores, *Fundamental Approaches to Software Engineering: 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, páginas 19–34. Springer Berlin Heidelberg, Berlin, Heidelberg.

Gong, X., Li, C. e Luo, Y. (2019). Intermediate data placement strategy for different data skew levels based on random sampling in spark. Em *Proceedings of the 2019 4th International Conference on Big Data and Computing*, ICBDC 2019, páginas 17–23, New York, NY, USA. ACM.

Guha, S., Li, Y. e Zhang, Q. (2017). Distributed partial clustering. Em *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*, página 143–152, New York, NY, USA. Association for Computing Machinery.

Gulati, S. e Kumar, S. (2017). *Apache Spark 2.x for Java Developers*, volume 1. Packt Publishing Ltd.

Hartmann, T., Moawad, A., Fouquet, F., Nain, G., Klein, J. e Le Traon, Y. (2015). Stream my models: Reactive peer-to-peer distributed models@run.time. Em *ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, páginas 80–89.

Hermann, F., Ehrig, H., Golas, U. e Orejas, F. (2014). Formal analysis of model transformations based on triple graph grammars. *Mathematical Structures in Computer Science*, 24(4).

Hili, N. (2016). A metamodeling framework for promoting flexibility and creativity over strict model conformance. Em *Proceedings of the 2nd Workshop on Flexible Model Driven Engineering co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2016), Saint-Malo, France, October 2, 2016*, páginas 2–11.

Imre, G. e Mezei, G. (2012). Parallel graph transformations on multicore systems. Em *Proceedings of the 2012 International Conference on Multicore Software Engineering, Performance, and Tools*, páginas 86–89. Springer-Verlag.

Jackson, D. e Torlak, E. (2018). Alloy, 5.0/2018. `http://alloytools.org/download/`. Acessado em 09/2019.

Jakumeit, E., Buchwald, S. e Kroll, M. (2010). Grgen.net. *International Journal on Software Tools for Technology Transfer*, 12(3):263–271.

Jouault, F., Allilaire, F., Bézivin, J. e Kurtev, I. (2008). Atl: A model transformation tool. *Science of Computer Programming*, 72(1–2):31 – 39. Special Issue on Second issue of experimental software and toolkits (EST).

Jouault, F. e Bézivin, J. (2006). Km3: A dsl for metamodel specification. Em Gorrieri, R. e Wehrheim, H., editores, *Formal Methods for Open Object-Based Distributed Systems*, páginas 171–185, Berlin, Heidelberg. Springer Berlin Heidelberg.

Jouault, F. e Kurtev, I. (2005). Transforming models with atl. Em *Proceedings of the 2005 International Conference on Satellite Events at the MoDELS*, MoDELS'05, páginas 128–138, Berlin, Heidelberg. Springer-Verlag.

Kahani, N., Bagherzadeh, M., Cordy, J. R., Dingel, J. e Varró, D. (2019). Survey and classification of model transformation tools. *Softw. Syst. Model.*, 18(4):2361–2397.

Kambatla, K., Kollias, G., Kumar, V. e Grama, A. (2014). Trends in big data analytics. *Journal of Parallel and Distributed Computing*, 74(7):2561 – 2573. Special Issue on Perspectives on Parallel and Distributed Processing.

Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W. e Wimmer, M. (2012). Model transformation by-example: A survey of the first wave. Em *Conceptual Modelling and Its Theoretical Foundations*, páginas 197–215. Springer-Verlag.

Karau, H. e Warren, R. (2017). *High Performance Spark*, volume 1. Ó Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA, USA, 1 ed. edition.

Kent, S. (2002). Model driven engineering. Em Butler, M., Petre, L. e Sere, K., editores, *Model-Driven Engineering Languages and Systems*, páginas 286–298. Springer Berlin Heidelberg.

Kessentini, M., Wimmer, M., Sahraoui, H. e Boukadoum, M. (2010). Generating transformation rules from examples for behavioral models. Em *Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications*, BM-FA '10, páginas 1–7, New York, NY, USA. ACM.

Khalek, S. A. (2011). *Systematic Testing Using Test Summaries: Effective and Efficient Testing of Relational Applications*. Tese de doutorado, University of Texas, Austin, USA.

Khronos, G. O. (2019). Opencl, release 2.2. `https://www.khronos.org/opencl/`. Online, acesssado 11/2019.

Kim, Y., Venkataramani, S., Chandrachoodan, N. e Raghunathan, A. (2019). Data subsetting: A data-centric approach to approximate computing. Em *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, páginas 576–581.

Kleiner, M., Didonet, D. F. M. e Santos, D. Q. (2013). Transformation as search. Em Van Gorp, P., Ritter, T. e Rose, L. M., editores, *Modelling Foundations and Applications*, volume 7949, páginas 54–69, Berlin, Heidelberg.

Klir, G. J. (2013). *Facets of systems science*, volume 7. Springer Science & Business Media.

Kolovos, D. S., Paige, R. F. e Polack, F. A. C. (2008). The epsilon transformation language. Em Vallecillo, A., Gray, J. e Pierantonio, A., editores, *Theory and Practice of Model Transformations: First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008 Proceedings*, páginas 46–60. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kougka, G., Gounaris, A. e Simitsis, A. (2018). The many faces of data-centric workflow optimization: a survey. *International Journal of Data Science and Analytics*, 6(2):81–107.

Lallchandani, J. T. e Mall, R. (2011). A dynamic slicing technique for uml architectural models. *IEEE Transactions on Software Engineering*, 37(6):737–771.

Lano, K. e Kolahdouz-Rahimi, S. (2010). Slicing of uml models using model transformations. Em *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems: Part II*, MODELS'10, páginas 228–242, Berlin, Heidelberg. Springer-Verlag.

Lano, K. e Kolahdouz-Rahimi, S. (2017). The uml-rsds toolset. `https://nms.kcl.ac.uk/kevin.lano/uml2web/`. Accessed in 2017/07.

Lano, K., Kolahdouz-Rahimi, S., Yassipour-Tehrani, S. e Sharbaf, M. (2018). A survey of model transformation design patterns in practice. *Journal of Systems and Software*, 140:48 – 73.

Lano, K. e Rahimi, S. K. (2011). Slicing techniques for uml models. *Journal of Object Technology*, 10(11):1–49.

Laptev, N., Mozafari, B., Mousavi, H., Thakkar, H., Wang, H., Zeng, K. e Zaniolo, C. (2016). Extending relational query languages for data streams. Em Garofalakis, M., Gehrke, J. e Rastogi, R., editores, *Data Stream Management: Processing High-Speed Data Streams*, páginas 361–386. Springer Berlin Heidelberg, Berlin, Heidelberg.

Li, J. (2008). *Window Queries Over Data Streams*. Tese de doutorado, Portland State University, Computer Science Department.

Liu, X. e Mellor-Crummey, J. (2013). A data-centric profiler for parallel programs. Em *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, páginas 28:1–28:12, New York, NY, USA. ACM.

Macedo, N. e Cunha, A. (2013). Implementing qvt-r bidirectional model transformations using alloy. Em *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, páginas 297–311, Berlin, Heidelberg.

Macedo, N. e Cunha, A. (2016). Least-change bidirectional model transformation with qvt-r and atl. *Softw. Syst. Model.*, 15(3):783–810.

Macedo, N., Guimarães, T. e Cunha, A. (2013). Model repair and transformation with echo. Em *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, páginas 694–697.

Macedo, N. F. M. (2014). *A Relational Approach to Bidirectional Transformation*. Tese de doutorado, University of Minho, Escola de Engenharia, Universidade do Minho.

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

Mazumdar, S., Seybold, D., Kritikos, K. e Verginadis, Y. (2019). A survey on data storage and placement methodologies for cloud-big data ecosystem. *Journal of Big Data*, 6(1):15.

Mens, T. (2013). Model transformation: A survey of the state of the art. *Model-Driven Engineering for Distributed Real-Time Systems: MARTE Modeling, Model Transformations and their Usages*, páginas 1–19.

Mens, T. e Van Gorp, P. (2006). A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125 – 142.

Mezic, I., Fonoberov, A. V., Fonoberova, M. e T., S. (2019). Spectral complexity of directed graphs and application to structural decomposition. *Complexity*, 2019.

Muzaffar, A. W., Mir, S. R., Anwar, M. W. e Ashraf, A. (2017). Application of model driven engineering in cloud computing: A systematic literature review. Em *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, New York, NY, USA. Association for Computing Machinery.

OMG (2014). Ocl object constraint language, 2014/february v2.4. `http://www.omg.org/spec/OCL/`. Accessed in 2019/09.

OMG (2016a). Qvt query view transformation, formal/2016-06-03 v1.3. `http://www.omg.org/spec/QVT`. Accessed in 2019/09.

OMG, O. M. G. (2016b). Mof - meta-object facility / 2.5.1. `http://www.omg.org/spec/MOF/`. 2019-09-27.

Pagán, J. E., Cuadrado, J. S. e Molina, J. G. (2015). A repository for scalable model management. *Software. System Model*, 14(1):219–239.

Paige, R. F., Matragkas, N. e Rose, L. M. (2016). Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272 – 280.

Qin, Y., Sheng, Q. Z., Falkner, N. J., Dustdar, S., Wang, H. e Vasilakos, A. V. (2016). When things matter: A survey on data-centric internet of things. *Journal of Network and Computer Applications*, 64:137 – 153.

Rauber, T. e Rnger, G. (2013). *Parallel Programming: For Multicore and Cluster Systems*. Springer Publishing Company, Incorporated, 2nd edition.

Rocha, H. e Valente, M. (2011). How annotations are used in java: An empirical study. Em *SEKE 2011 - Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering*, páginas 426–431.

Rutle, A., Rossini, A., Lamo, Y. e Wolter, U. (2012). A formal approach to the specification and transformation of constraints in mde. *The Journal of Logic and Algebraic Programming*, 81(4):422 – 457.

Scheidgen, M., Zubow, A., J., F. e T.H., K. (2012). Automated and transparent model fragmentation for persisting large models. Em *Model Driven Engineering Languages and Systems (MODELS 2012)*, páginas 102–118.

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31.

Schürr, A. (1995). Specification of graph translators with triple graph grammars. Em *in Proc. of the 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science (WG '94), Herrsching (D*. Springer.

Shaikh, A., Wiil, U. K. e Memon, N. (2011). Evaluation of tools and slicing techniques for efficient verification of uml/ocl class diagrams. *Advanced Software Engineering*, 5:1–18.

Shao, B. e Li, Y. (2018). Parallel processing of graphs. Em Fletcher, G., Hidders, J. e Larriba-Pey, J. L., editores, *Graph Data Management: Fundamental Issues and Recent Developments*, páginas 143–162. Springer International Publishing.

Smolders, B. (2017). Towards a solution for improving the performance of model transformations in model-driven development. Dissertação de Mestrado, Utrecht University, Utrecht, Netherlands.

Stachowiak, H. (1973). *Allgemeine Modelltheorie*. Springer Verlag, Wien, New York.

Stahl, T., Voelter, M. e Czarnecki, K. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

Tang, M., Shao, S., Yang, W., Liang, Y., Yu, Y., Saha, B. e Hyun, D. (2019). Sac: A system for big data lineage tracking. Em *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, páginas 1964–1967.

Tang, Z., Zhang, X., Li, K. e Li, K. (2018). An intermediate data placement algorithm for load balancing in spark computing environment. *Future Generation Computer Systems*, 78:287 – 301.

Tehrani, S. Y., Zschaler, S. e Lano, K. (2016). Requirements engineering in model-transformation development: An interview-based study. Em *Proceedings of the 9th International Conference on Theory and Practice of Model Transformations - Volume 9765*, páginas 123–137, New York, NY, USA.

Thang, N. X., Zapf, M. e Geihs, K. (2011). Model driven development for data-centric sensor network applications. Em *Proceedings of the 9th International Conference on Advances in Mobile Computing and Multimedia*, páginas 194–197, New York, NY, USA. ACM.

Tisi, M., Martínez, S. e Choura, H. (2013). Parallel execution of atl transformation rules. Em *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems - Volume 8107*, páginas 656–672, New York, NY, USA. Springer-Verlag New York, Inc.

Tomaszek, S., Leblebici, E., Wang, L. e Schürr, A. (2018). Model-driven development of virtual network embedding algorithms with model transformation and linear optimization techniques. Em Schaefer, I., Karagiannis, D., Vogelsang, A., Méndez, D. e Seidl, C., editores, *Modellierung 2018*, páginas 39–54, Bonn. Gesellschaft für Informatik e.V.

Ujhelyi, Z., Horvath, A. e Varro, D. (2012). Dynamic backward slicing of model transformations. Em *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, páginas 1–10, Washington, DC, USA. IEEE Computer Society.

Uzuncaova, E. e Khurshid, S. (2007). Kato: A program slicing tool for declarative specifications. Em *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, páginas 767–770, Washington, DC, USA.

Vasata, D. (2018). *Ferramenta de Programação e Processamento para Execução de Aplicações com Grandes Quantidades de Dados em Ambientes Distribuídos*. Tese de doutorado, Escola Politécnica da Universidade de São Paulo, Departamento de Engenharia de Computação e Sistemas Digitais.

Venkata, M. G., Aderholdt, F. e Z., P. (2017). Sharp: Towards programming extreme-scale systems with hierarchical heterogeneous memory. Em *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, páginas 145–154.

Wagner, C. (2014). *Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems*. Vieweg Teubner Verlag, 1 edition.

Wang, H. e Abraham, Z. (2015). Concept drift detection for streaming data.

Wei, R., Kolovos, D. S., Garcia-Dominguez, A., Barmpis, K. e Paige, R. F. (2016). Partial loading of xmi models. Em *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, páginas 329–339, New York, NY, USA. ACM.

Weiser, M. (1981). Program slicing. Em *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, páginas 439–449.

Wilkinson, B. e Allen, M. (1999). *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Zeng, J. e Yu, H. (2018). A distributed infomap algorithm for scalable and high-quality community detection. Em *Proceedings of the 47th International Conference on Parallel Processing*, New York, NY, USA. Association for Computing Machinery.

Zhou, J. e Demsky, B. (2010). Bamboo: A data-centric, object-oriented approach to many-core software. *SIGPLAN Not.*, 45(6):388–399.

Zomaya, A. Y. e Sakr, S. (2017). *Handbook of Big Data Technologies*, volume 1. Springer International Publishing, 1 ed. edition.

Zu, X., Bai, Y. e Yao, X. (2016). Data-centric publish-subscribe approach for distributed complex event processing deployment in smart grid internet of things. Em *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, páginas 710–713.

APÊNDICE A – REGRAS DE TRANSFORMAÇÃO DE MODELOS

Neste apêndice, estão as regras de transformação de modelos especificadas para a Dc4MT e experimentadas nos ambientes local e distribuído. O conjunto completo contendo os filtros e as regras implementado para a Dc4MT está em: `https://github.com/lzcamargo/Dc4MT`. Na Listagem A.1, são apresentadas as regras de transformação `Families2Persons`.

Listagem A.1: Regras `Families2Persons`

```
1   val personMale = lstNmFamilies.select($"src".alias("origem"), $"value".alias("lastName"))
2   .join(fstLstNmMale, $"src" === $"origem")
3   .select(concat($"lastName", lit(" "), $"value") alias "fullName")
4
5   val personFemale = lstNmFamilies.select($"src".alias("origem"), $"value".alias("lastName"))
6     .join(fstLstNmFemale, $"src"=== $"origem")
7     .select(concat($"lastName", lit(" "), $"value") alias "fullName")
8
9   // Persons of Gender union in a partition
10  val personsDF = personMale.withColumn("Gender", lit("Male"))
11    .union(personFemale.withColumn("Gender",lit("Female")))
12    .coalesce(1)
```

O domínio das regras de transformação `Class2Relational` foi retirado da atlTransformations[1] e especificado em Scala para transformar os modelos Class, representados em grafos (GraphFrames), em modelos Relational. Na Listagem A.2, estão as regras de transformação `Class2Relational`.

Listagem A.2: Regras `Class2Relational`

```
1  // ............... ClassDataTypes2RelationalDataTypes ...........
2   val relationalDtTypeDF = lnkDataType.join(vertices, $"id" === $"dst")
3     .select($"idDtType", $"value")
4
5  // ............... Package2Schema Rule ..........................
6   val schemaDF = srcDstPackages.join(vertices, $"dst" === $"id")
7     .select($"src", $"value".alias("schema"))
8
9  // ............... Class2Table Rule .................................
10  val lnkSchClas = schemaDF.select($"src".alias("origem")).join(persistClassSrc)
11    .filter($"origem" === $"src").select($"src", $"dst")
12  val tableDF = lnkSchClas.select($"src", $"dst").join(persistClassName,
13    $"origem" === $"dst").select($"src", $"value".alias("name"), $"dst")
14  val objColumnDF = tableDF.select($"dst", lit("objectId")
15    .alias("name"),lit(typeId).alias("type"))
16
17 // ................ SingleValuedDataTypeAttribute2Column Rule ...........
18  val dataTypeColumnDF = simpleAtt.join(attNameType, $"src" === $"origem")
19   .select($"origem", $"name", $"dstAttType")
20
21 // ................ ClassAttribute2Column Rule .................
22  val clasAttColumnDF = attTypeClasName
23   .select($"origem", concat($"name", lit("Id"))
24   .alias("name"), lit(typeId).alias("type"))
25
26 // ................. MultiValuedDataTypeAttribute2Column Rule .............
27  val multDataTypeTable = multClassAttName
28   .select($"origem", concat($"value", lit("_"), $"attName").alias("name"))
29  val multValdDataTypeColumnId = multClassAttName
30   .select($"origem", concat($"attName", lit("Id")).alias("name"),lit(typeId).alias("type"))
31  val multValdDataTypeColumnDF = multClassAttName.select($"origem", $"attName"
32   .alias("name"), $"dstAtttype")
33
```

[1]https://www.eclipse.org/atl/atlTransformations/#Class2Relational

```
34  // ................. MultiValuedClassAttribute2Column Rule ................
35    val assAttName = srcAttName.join(dstAttName,$"origem" === $"src")
36    val assTable = assAttName.select($"origem", concat($"name",
37      lit("_"), $"nameDst").alias("name"))
38    val multValdColumnId = assAttName.select($"origem", concat($"name",
39      lit("Id")).alias("name"), lit(typeId).alias("type"))
40    val multValdCollumnFk = assAttName.select($"origem", concat($"nameDst",
41      lit("Id")).alias("name"), lit(typeId).alias("type")).distinct()
```

As Regras `IMDb2Identity` e `IMBb2FindCouple` são apresentadas nas Listagens A.3 e A.4, o domínio dessas regras foi obtido do Projeto Lintra[2] e implementado na linguagem Scala.

Listagem A.3: Regras `IMDb2Identy`

```
1  // .............. Movie Rule ................
2    val movies = moviesId.join( vertices, $"dst" === $"id")
3      .select($"movid", $"key", $"value")
4
5  // ............. Actor Rule .......................
6    val actorName = actorId.filter($"key" === "name").join(vertices, $"dst" === $"id")
7      .select($"actid".alias("actidn"), $"value".alias("name"))
8    val actorMovie = actorId.filter($"key" === "movies")
9      .select($"actid", $"dst", $"key".alias("movies")).distinct()
10
11 //....... Actress Rule ...........
12   val actressName = actressId.filter($"key" === "name").join(vertices, $"dst" === $"id")
13     .select($"actid".alias("actidn"), $"value".alias("name"))
14   val actressMovie = actressId.filter($"key" === "movies")
15     .select($"actid", $"dst", $"key".alias("movies")).distinct()
```

Listagem A.4: Regras `IMDb2findCouple`

```
1  // ........ Main Actor Rule ....................
2  val mainActorMovie = actorInMovies.select($"src".alias("origem"))
3    .join(actorId,$"src" === $"srcm")
4    .select($"origem", $"dst".alias("destino"))
5
6  // ............ People1 of the Couple Rule....................
7  val people1DF = coupleId.join( edges.filter($"key" === "name"), $"origem" === $"src")
8    .select($"origem", $"dst").distinct()
9  val people1Name = people1DF.join( vertices, $"dst" === $"id")
10   .select($"origem", $"value".alias("name"))
11
12 // ............ People2 of the Couple Rule....................
13 val people2DF = coupleId.join( edges.filter($"key" === "name"), $"destino" === $"dst")
14   .select($"origem", $"dst").distinct()
15 val people2Name = people2DF.join( vertices, $"dst" === $"id")
16   .select($"origem", $"value".alias("name"))
17
18 // ............ Union of the People1 and People2 ....................
19 val couplesDF = people1Name.union(people2Name)
```

Na Listagem A.5, estão as três regras de transformação `DBLP2ICMT`, cujo domínio também é do Projeto Lintra.

Listagem A.5: Regras `DBLP2ICMT`

```
1  // ...... DBLP2ICMT1 Rule ........................
2  val icmtAuthor = icmtAutorId.join(edges.filter($"key" === "authors"), $"origem" === $"src")
3    .select($"origem", $"dst")
4  val icmtAuthorQtde = icmtAuthor.groupBy($"dst").count()
5  val icmtAuthorsQtdeDF = icmtAuthorQtde.join(vertices, $"dst" === $"id")
6    .select($"value", $"count".alias("Qtde")).distinct()
7
8  // ...... DBLP2ICMT2 Rule ........................
9  val authorActiveDF = icmtAuthorsActive.join(edges
10   .filter($"key" === "name" ), $"origem" === $"src").select($"dst")
```

[2]https://atenea.lcc.uma.es/projects/LinTra.html

```
11     .join(vertices, $"dst" === $"id")
12     .select($"value").withColumn("Active", lit("yes")).distinct()
13    val authorInactiveDF = icmtAuthorsActive.join(edges
14      .filter($"key" === "name" ), $"origem" === $"src").select($"dst")
15      .join(vertices, $"dst" === $"id")
16      .select($"value").withColumn("Active", lit("no")).distinct()
17
18   // ...... DBLP2ICMT3 Rule ........................
19   val authorPubYear = icmtAuthorsInactive.join(edges.filter($"key" === "year"),
20    $"origem" === $"src").select($"origem", $"dst")
21   val authorPubReturn = authorPubYear
22     .join(vertices.when($"value".geq('yerBase - 5)), $"dst" === $"id")
23     .select($"origem")
24   val authorReturnNameDF = authorPubReturn.join(edges
25     .filter($"key" === "name" ), $"origem" === $"src").select($"dst")
26     .join(vertices, $"dst" === $"id").select($"value").distinct()
```