API em Python para Extração de Dados de Proveniência usando o DfAnalyzes

Vinícius Silva Campos[1]

[1]Escola Politécnica – Universidade Federal do Rio de Janeiro (UFRJ)

vinicius.s.campos@poli.ufrj.br

1. Motivação

Experimentos científicos computacionais tem se tornado cada vez mais comuns e desafiadores. A complexidade, o número de execuções e o volume de dados produzidos por esses experimentos tem aumentado cada vez mais tornando-os problemas difíceis na área de *Big Data*. Diferentes são as maneiras de se modelar a execução dessas simulações computacionais. Dentre elas, temos a abstração de *workflows* científicos que, pode ser vista como o encadeamento de tarefas (programas) responsáveis por realizar transformações sob os dados fornecidos como entrada gerando, portanto, um fluxo de dados entre essas tarefas.

Ao longo da execução de uma simulação computacional dados são gerados e consumidos pelas tarefas (programas) a esses dados damos o nome de dados de proveniência. Esses dados são importantes, pois favorecem a reprodutibilidade do experimento, aumentam a confiabilidade e auxiliam, por exemplo, na verificação de em qual ponto a solução passou a divergir. Portanto, torna-se desejável no atual cenário de *Big Data*, realizar a captura desses dados de proveniência de forma adequada visando não comprometer o tempo de execução do experimento.

Porém, a área científica já possui uma gama de soluções computacionais bem estabelecidas para cada área de domínio. A área de bioinformática, por exemplo, possui uma grande quantidade de *scripts* desenvolvidos na linguagem de programação *Python* e, portanto, não faria sentido solicitar que estes *scripts* fossem adaptados para linguagens declarativas específicas para que o gerenciamento de sua execução seja facilitado. Dessa forma, seria inviável solicitar que estes cientistas reescrevessem seus programas para que conseguissem passar a capturar os dados de proveniência e seria igualmente improvável desenvolver uma ferramenta genérica o suficiente afim de contemplar todos os possíveis modelos de dados para as mais diversas áreas cientificas.

2. Proposta

Uma API (*Application Programming Interface*) na linguagem de programação *Python*. Por meio dessa API, o cientista pode instrumentar seu código, já desenvolvido, seja por meio da incorporação do código da API em sua solução ou por meio da utilização de RPC (*Remote Procedure Call*). Essa API funcionará como um *wrapper* criando uma camada de abstração,reduzindo a complexidade demandada para o cientista instrumentar seu código para permitir a captura de dados de proveniência e possui integração com a API Restful do *DfAnalyzer* por meio de requisições HTTP do tipo *POST* para realizar a ingestão dos dados de proveniência na base.

A escolha pela implementação em *Python*, ocorreu devido a grande adesão que esta linguagem obteve, sobretudo em áreas de domínio de cientistas computacionais como

os bioinformatas e em áreas relacionadas à *data science* e *Machine Learning (TensorFlow)*.

Essa API permitirá que os cientistas capturem dados de proveniência prospectiva e retrospectiva de forma intuitiva visando abstrair a complexidade do processo para o cientista. A proveniência prospectiva, como o nome sugere, é obtida antes mesmo do início da execução do *workflow* e é responsável por definir o fluxo de dados ao longo do experimento. A retrospectiva, por sua vez, é o produto da execução do *workflow* contendo informações sobre os dados gerados e consumidos ao longo da execução, que podem contemplar dados de performance, domínio, arquivos consumidos, etc.

3. Referencial Teórico

Com o intuito de facilitar as discussões sobre a solução desenvolvida, que é uma extensão do *DfAnalyzer*, esta seção apresenta a abordagem álgébrica para a composição de *workflows* científicos, o modelo de dados de proveniência proveniência para gerência da execução e o *Dataflow Analyzer*.

3.1. SciWfA: Álgebra Relacional de *Workflows* científicos

A abordagem algébrica é uma das formas de descrição da composição de workflows científicos. Tal abordagem fornece informações que impactam diretamente na estratégia de execução adotada pelo SGWfC, uma vez que apresenta o encadeamento lógico dos programas científicos na composição de um workflow científico. Tendo como principais contribuições a composição e o potencial de otimização do plano de execução de workflows científicos, [Ogasawara et al. 2011] apresentaram uma álgebra para workflows científicos, conhecida como Scientific Workflow Algebra (SciWfA), que é baseada na álgebra relacional de banco de dados.

Para descrever os workflows científicos usando a SciWfA, as relações são utilizadas para representar um conjunto de dados produzido ou consumido pelas atividades. Essa abordagem permite a representação uniforme de dados ao longo do workflow. Como na álgebra relacional, as relações são definidas por um conjunto de tuplas. Essa álgebra estende a álgebra relacional [F. Codd 1990] através da introdução de um conjunto de novos operadores algébricos que utilizam tanto atividades quanto relações como operandos. Esses operadores adicionais atuam nas atividades do workflow por meio da razão entre o número de tuplas de entrada consumidas e de saída produzidas. Por meio dos operadores algébricos das relações e das atividades, é possível representar expressões algébricas [F. Codd 1990]. A Figura 1 apresenta de forma genérica a expressão algébrica de uma atividade na SciWfA. Essa álgebra apresenta seis operadores algébricos para a especificação de atividades, conhecidas como Map, Reduce, SplitMap, Filter, MRQuery e SRQuery. Esses operadores diferem entre si pelo número de tuplas de entrada consumidas e de saída produzidas, como pode ser observada na Figura 2.

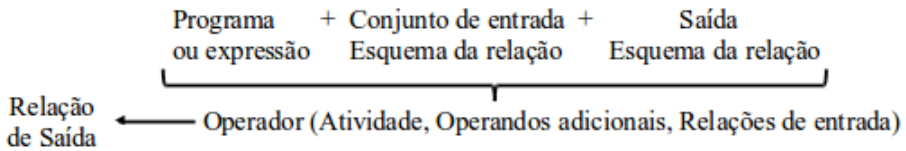

Figura 1. Expressão algébrica de uma atividade na SciWfa.

Operador	Tipos de atividades operadas	Operandos adicionais	Relação de tuplas consumidas / produzidas
Map	Programa	Relação	1:1
SplitMap	Programa	Referência à arquivo, relação	1:m
Reduce	Programa	Conjunto de atributos de agrupamento, relação	n:1
Filter	Programa	Relação	1:(0-1)
SRQuery	Expressão da álgebra relacional	Relação	n:m
MRQuery	Expressão da álgebra relacional	Conjunto de relações	n:m

Figura 2. Operadores algébricos

3.2. Modelo de dados de proveniência PROV-Df para gerência da execução de *workflows* científicos

Como mencionado anteriormente, alguns SGWfC proveem a análise de proveniência em tempo de execução, permitindo o monitoramento do workflow, os ajustes finos em valores de atributos durante a condução da simulação e a possibilidade de interação nas execuções do workflow [Mattoso et al. 2014]. Por meio da análise dos dados de proveniência [Costa et al. 2013], é possível também examinar a qualidade de um resultado baseado nos dados de entrada e nas atividades do workflow, rastrear o fluxo de dados envolvido em uma simulação computacional ou mesmo monitorar comportamentos inesperados em tempo de execução. As propagações de dados entre atividades do workflow (i.e., invocação de um programa científico) descrevem um fluxo de dados para as simulações computacionais. Através da análise desse fluxo de dados, é possível entender a estrutura do workflow como um todo, assim como investigar comportamentos específicos do domínio do experimento científico.

Alguns SGWfC permitem o registro de dados associados à estrutura do workflow (i.e., proveniência prospectiva), à execução dos workflows (i.e., proveniência retrospectiva) e dados de desempenho de execução. Diferentemente dos outros trabalhos relacionados, apenas a nossa proposta considera uma base de dados de proveniência enriquecida com dados de domínio, capturando e armazenando tais dados em tempo de execução. A presença dos dados de desempenho de execução e dos dados de domínio na base de dados de proveniência torna possível também a especificação de consultas com o intuito de analisar, por exemplo, a eficiência de modelos computacionais ou a ocorrência de erros em função do tempo de execução de uma programa científico.

Com o intuito de padronizar a representação da base de proveniência empregada em sistemas de workflows científicos, modelos de dados têm sido propostos para a apoiar a gerência de dados de proveniência em workflows científicos. Dentre os modelos de dados de proveniência propostos temos o PROV-Df [Silva et al. 2016]. O PROV-Df permite armazenar os dados de proveniência em tempo de execução, ao mesmo tempo em que auxilia na distribuição das tarefas para execução paralela, provendo vantagens como a redução do tempo de execução, a possibilidade de utilização mais adequada dos recursos disponíveis em ambientes de PAD e a expansão da capacidade de execução de workflows científicos em larga escala. Vale ressaltar que tal modelo também permite a gerência do fluxo de dados gerado durante a execução de workflows científicos, relacionando elementos de dados entre diferentes atividades do workflow.

3.3. Arquitetura ARMFUL

A Arquitetura ARMFUL permite a análise de dados brutos a partir de múltiplos arquivos. Realiza isso através de análises de dados científicos por meio da recuperação de dados frutos de arquivos e os relacionando ao longo do fluxo de dados de proveniência [Silva et al. 2017]. A gerência de dados brutos exige funcionalidades que são específicas do domínio, portanto, modelos de dados de proveniência deixam essa representação na granularidade grande. Isso exige muito esforço por parte dos usuários dos SWMS em desenvolver,para cada domínio, modelos de dados e programas específicos para acessar, extrair e relacionar dados de domínio com os dados de proveniência. A arquiterura ARMFUL tem como objetivo auxiliar nesse esforço por meio da representação de alguns componentes genéricos que modelam e relacionam dados específicos de domínio à proveniência na mesma base [Silva et al. 2017].

3.4. *DfAnalyzer: DataflowAnalyzer*

O *DfAnalyzer* é uma ferramenta genérica para a extração de dados de proveniência e dados científicos em simulações computacionais intensivas em dados. O *DfAnalyzer* é uma instância da arquitetura ARMFUL apresentada em 3.3, e sua arquitetura está representada na Figura 3.

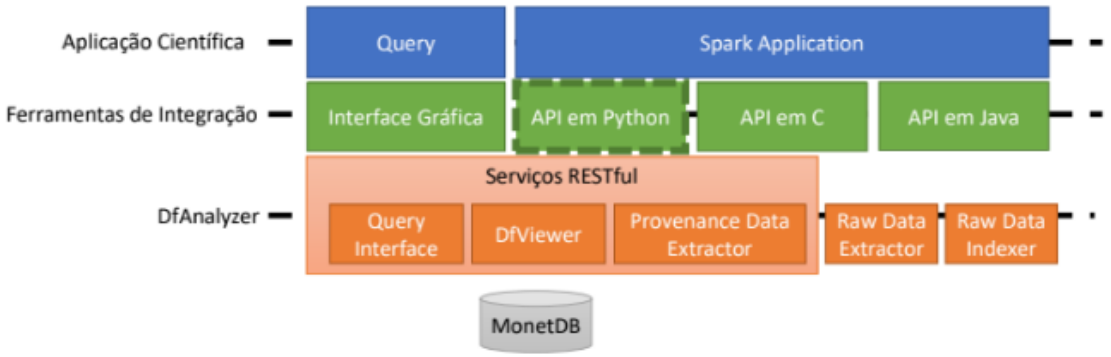

Figura 3. Arquitetura *DfAnalyzer*

4. Desenvolvimento

Conforme apresentado em 2, foi decidido pela implementação de uma API na linguagem de programação *Python* para possibilitar a extração de dados de proveniência de experimentos computacionais por meio da instrumentação dos códigos das aplicações científicas. Essa API se comporta como um *Wrapper* que viabiliza a integração do código em *Python* com a solução Restful do *DfAnalyzer* disponibilizada em [Repositório b]. A intrumentação do código para extração de dados de proveniência já era possível por meio da utilização do PDG (*Provenance Data Gatherer*) do *DfAnalyzer*. Porém, a solução fornecia suporte apenas para a linguagem de programação *Java*. Portanto, seria necessário implementar chamadas RPC (*Remote Procedure Call*) ao longo do código da aplicação científica caso a mesma tivesse implementada em *Python*, por exemplo.

A estrutura da solução desenvolvida foi dividida em três vertentes sendo elas: Proveniência prospectiva, Proveniência Retrospectiva e PDE (em *Python*). Essas implementações dependem das classes base *ProvenanceObject* e PDE e serão descritas abaixo.

4.1. *ProvenanceObject*

O ProvenanceObject, como o nome sugere, possui as propriedades básicas que a solução assume que um objeto de proveniência deveria, essas propriedades são: *tag* e uma representação em *json*.

```
class ProvenanceObject(object):

    def __init__(self, tag):
        self._tag = tag
        self._json = {}

    def __repr__(self):
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'tag': self._tag}
        return self._json
```

4.2. PDE (*Provenance Data Extractor*)

O PDE é responsável pelas etapas de extração de dados brutos (*Raw Data Extractor*) e pela indexação de dados brutos (*Raw Data Indexer*) presentes na Figura 3.

A classe PDE implementada utiliza-se da abstração do tipo *factory* para permitir a instanciação, estaticamente, de classes responsáveis por encapsular os dados de proveniência (program, set, dataflow, etc). Além disso, possui também dois métodos responsáveis por realizar a ingestão dos dados por meio da integração com a API Restful do *DfAnalyzer*, são eles: `ingest_dataflow_json e ingest_task_json`.

```
class PDE(object):
    def __init__(self, configuration=None):
        self._configuration = configuration

    # region ProspectiveProvenance
    @staticmethod
    def dataflow(tag):
        return Dataflow(tag)

    @staticmethod
    def program(name, path):
        return Program(name, path)

    @staticmethod
    def set(tag, type, attributes=None, extractors=None,
     ↪ extractor_combinations=None):
        return Set(tag, type, attributes, extractors,
         ↪ extractor_combinations)
```

```
    @staticmethod
    def extractor(tag, cartridge=None, extension=None):
        return Extractor(tag, cartridge, extension)

    @staticmethod
    def transformation(tag, programs=None, sets=None):
        return Transformation(tag, programs, sets)

    @staticmethod
    def attribute(tag, type, extractor=None):
        return Attribute(tag, type, extractor)

    @staticmethod
    def set_type():
        return SetType

    @staticmethod
    def attribute_type():
        return AttributeType

    @staticmethod
    def extractor_cartridge():
        return ExtractorCartridge

    @staticmethod
    def extractor_extension():
        return ExtractorExtension

    # endregion

    # region RetrospectiveProvenance
    @staticmethod
    def task(tag, dataflow_tag, transformation_tag,
     ↪ resource, workspace, status, id, performances=[],
             output=None, error=None, sub_id=None,
              ↪ invocation=None, dependency=None,
              ↪ files=[], idatasets=[]):
        return Task(tag, dataflow_tag, transformation_tag,
         ↪ resource, workspace, status, id, performances,
         ↪ output, error, sub_id,
                     invocation, dependency, files,
                      ↪ idatasets)

    @staticmethod
    def file(name, path):
        return File(name, path)
```

```
    @staticmethod
    def performance(tag, method_type, start_time,
     ↪  task_id=None, description=None, invocation=None
                    , task_sub_id=None, end_time=None):
        return Performance(tag, description, method_type,
         ↪  invocation, task_id, task_sub_id, start_time,
         ↪  end_time)

    @staticmethod
    def element(tag, values):
        return Element(tag, values)

    @staticmethod
    def idataset(tag, set, elements):
        return IDataset(tag, set, elements)

    @staticmethod
    def data_collection(idataset, tag=None):
        return DataCollection(idataset, tag)

    @staticmethod
    def status_type():
        return StatusType

    @staticmethod
    def method_type():
        return MethodType

    # endregion
    # region RestAPI
    @staticmethod
    def ingest_dataflow_json(base_url, dataflow_json):
        url = base_url + '/pde/dataflow/json'
        r = requests.post(url, json=dataflow_json)
        print(r.status_code)
        print(r.content)

    @staticmethod
    def ingest_task_json(base_url, task_json):
        url = base_url + '/pde/task/json'
        r = requests.post(url, json=task_json)
        print(r.status_code)
        print(r.content)

    # endregion
```

4.3. Proveniência Prospectiva

A captura de dados de proveniência prospectiva foi feita por meio da criação de classes para representar cada propriedade do modelo de proveniência proposto pelo *DfAnalyzer*. As classes criadas foram: Attribute, AttributeType, Dataflow, Extractor, ExtractorCartridge, ExtractorCombination, ExtractorExtension, Program, Set, SetType e Transformation. Cada uma das classes será apresentada individualmente a seguir.

4.3.1. Attribute

Essa classe é responsável por encapsular as informações referentes aos atributos dos *Datasets*. Cada atributo possui uma *tag*, um tipo e um extrator atrelado.

```
class Attribute(ProvenanceObject):

    def __init__(self, tag, type, extractor = None):
        ProvenanceObject.__init__(self,tag)
        self._type = type
        self._extractor = extractor
        self._json = {}

    def __repr__(self):
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'name': self._tag, 'type':
         ↪  self._type}
        if(self._extractor!=None):
            self._json['extractor'] = self._extractor
        return self._json
```

4.3.2. AttributeType

Essa classe é responsável por definir os possíveis tipos de um *Attribute*, sendo apenas um Enum.

```
from enum import Enum

class AttributeType(Enum):
    TEXT = 'TEXT'
    NUMERIC = 'NUMERIC'
    FILE = 'FILE'
```

4.3.3. Extractor

Essa classe é responsável por encapsular as informações referentes aos extratores dos dados. Um extrator é definido por uma *tag*, um ”cartucho”(*cartridge*) e uma extensão (ex.: CSV). Para ser criado basta que seja fornecida uma *tag*, portanto, poderíamos criar previamente todos os extratores que serão utilizados ao longo do *dataflow* e depois associar um tipo de cartucho e uma extensão à eles.

```
class Extractor(ProvenanceObject):

    def __init__(self, tag, cartridge = None , extension=
     ↪ None):
        ProvenanceObject.__init__(self,tag)
        self._cartridge = cartridge
        self._extension = extension
        self._json = {}

    def __repr__(self):
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'tag': self._tag}
        if(self._cartridge!=None):
            self._json['cartridge'] = self._cartridge
        if (self._extension != None):
            self._json['extension'] = self._extension
        return self._json
```

4.3.4. ExtractorCartridge

Essa classe é responsável por encapsular as informações referentes aos possíveis tipos para cartuchos para os extratores. Conforme esperado, sua implementação é bastante simples, sendo apenas um Enum que define os possíveis valores de cartucho.

```
from enum import Enum

class ExtractorCartridge(Enum):
    EXTRACTION = 'EXTRACTION'
    INDEXING = 'INDEXING'
```

4.3.5. ExtractorExtension

Essa classe é responsável por encapsular as informações referentes às possíveis extensões para os extratores. Conforme esperado, sua implementação é bastante simples, sendo apenas um Enum que define os possíveis valores das extensões.

```
from enum import Enum

class ExtractorExtension(Enum):
    PROGRAM = 'PROGRAM'
    CSV = 'CSV'
    FITS = 'FITS'
    FASTBIT = 'FASTBIT'
```

4.3.6. ExtractorCombination

Essa classe, como o nome sugere, é responsável por representar o cenário em que um ou mais estratores são combinados. Essas combinações são armazenadas como podendo ser do tipo *inner* e *outer*. Dessa forma, por exemplo, podemos ter dois extratores como *inner* e um outro extrator como *outer* no processo de extração de dados de proveniência.

```
class ExtractorCombination(ProvenanceObject):

    def __init__(self, outer, inner,tag = ""):
        ProvenanceObject.__init__(self,tag)
        self._outer = outer
        self._inner = inner
        self._json = {}

    def __repr__(self):
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'tag': self._tag, 'outer':
         ↪ self._outer, 'inner': self._inner}
        return self._json
```

4.3.7. Program

Essa classe é responsável por encapsular as informações referentes aos programas utilizados ao longo do *dataflow*. Um programa é definido pela sua *tag* e o seu *path*, sendo o fornecimento dessas propriedades obrigatório ao instanciar um objeto da classe.

```
class Program(ProvenanceObject):
    def __init__(self, tag, path):
        ProvenanceObject.__init__(self, tag)
        self._path = path
        self._json = {}

    def __repr__(self):
```

```
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'name': self._tag, 'path':
         ↪ self._path}
        return self._json
```

4.3.8. Set

Essa classe é responsável por encapsular as informações referentes aos *datasets* gerados e consumidos ao longo da execução do *dataflow*. As propriedades obrigatórias no momento de sua criação são a *tag* e o tipo. Porém, um *Set* (*DataSet*) é definido pelos atributos que o compõe, os extratores que operam sobre ele e, possivelmente, sobre a combinação de extratores.

```
class Set(ProvenanceObject):
    def __init__(self, tag, type, attributes=None,
     ↪ extractors=None, extractor_combinations=None):
        ProvenanceObject.__init__(self, tag)
        self._type = type
        self._attributes = attributes
        self._extractors = extractors
        self._extractor_combinations =
         ↪ extractor_combinations
        self._json = {}

    def __repr__(self):
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'tag': self._tag, 'type':
         ↪ self._type}
        if (self._attributes != None):
            self._json['attributes'] = self._attributes
        if (self._extractors != None):
            self._json['extractors'] = self._extractors
        if (self._extractor_combinations != None):
            self._json['extractor.combination'] =
             ↪ self._extractor_combinations
        return self._json
```

4.3.9. SetType

Essa classe é responsável por definir os possíveis tipos de um *DataSet*, sendo apenas um Enum.

```
from enum import Enum

class SetType(Enum):
    INPUT = 'INPUT'
    OUTPUT = 'OUTPUT'
```

4.3.10. Transformation

Essa classe é a maior granularidade dentro de um *dataflow*. Possui uma *tag*, programas e *datasets* atrelados à ela. Porém, apenas a *tag* é um atributo obrigatório no momento de criação. Dessa forma, pode-se instrumentar o código do cientista numa abordagem *top-down* criando todos os elementos que são mais alto nível antes.

```
class Transformation(ProvenanceObject):
    def __init__(self, tag, programs=None, sets=None):
        ProvenanceObject.__init__(self, tag)
        self._programs = programs
        self._sets = sets
        self._json = {}

    def __repr__(self):
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'tag': self._tag}
        if (self._programs != None):
            self._json['programs'] = self._programs
        if (self._sets != None):
            self._json['sets'] = self._sets
        return self._json
```

4.3.11. Dataflow

Essa classe é a principal classe da proveniência prospectiva, sendo responsável por encapsular todas as propriedades necessárias para definir um *dataflow* segundo o modelo de dados de proveniência seguido pelo *DfAnalyzer*. Seus atributos são *tag* (nome do *dataflow*) e *transformations* (conjunto de transformações que compõe esse *dataflow*).

```
class Dataflow(ProvenanceObject):

    def __init__(self, tag, transformations = None):
        ProvenanceObject.__init__(self,tag)
        self._transformations = transformations
        self._json = {}
```

```
    def __repr__(self):
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'tag': self._tag}
        if(self._transformations!=None):
            self._json['transformations'] =
             ↪ self._transformations
        return self._json
```

4.4. Proveniência Retrospectiva

Tal como em 4.3, a captura de dados de proveniência retrospectiva foi feita por meio da criação de classes para representar cada propriedade do modelo de proveniência proposto pelo *DfAnalyzer*. As classes criadas foram: DataCollection, Element, ExecuteDataflow, File, IDataset, MethodType, Performance, StatusType e Task. Cada uma das classes será apresentada individualmente a seguir.

4.4.1. Element

Essa classe é responsável por encapsular valores aos elementos (dados), gerados e consumidos, ao longo da execução do *dataflow*. Cada elemento está atrelado à um ou mais *IDataset*.

```
class Element(ProvenanceObject):

    def __init__(self, tag, values):
        ProvenanceObject.__init__(self,tag)
        self._values = values
        self._json = {}

    def __repr__(self):
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'tag': self._tag, 'values':
         ↪ self._values}
        return self._json
```

4.4.2. IDataset

Essa classe é responsável por encapsular os *data elements* (elementos de dados) gerados ao longo da execução de um *dataflow* com o *dataset* ao qual pertencem.

```
class IDataset(ProvenanceObject):
    def __init__(self, tag, set, elements):
```

```
        ProvenanceObject.__init__(self, tag)
        self._set = set
        self._elements = elements
        self._json = {}

    def __repr__(self):
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'tag': self._tag, 'set': self._set,
         ↪ 'elements': self._elements}
        return self._json
```

4.4.3. ExecuteDataflow

Essa classe é responsável por encapsular as informações referentes à execução de um dado *dataflow*. Possui como atributos uma *tag* (de execução), uma versão e um identificador para o *dataflow* que está sendo instanciado para a execução.

```
class ExecuteDataflow(ProvenanceObject):

    def __init__(self, tag, version, dataflow):
        ProvenanceObject.__init__(self,tag)
        self._version = version
        self._dataflow = dataflow
        self._json = {}

    def __repr__(self):
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'tag': self._tag, 'version':
         ↪ self._version, 'dataflow': self._dataflow}
        return self._json
```

4.4.4. File

Essa classe é responsável por encapsular as informações referentes aos arquivos gerados e consumidos ao longo da execução do *dataflow*. Possui como atributos uma *tag* e um *path* que indica onde o arquivo está.

```
class File(ProvenanceObject):
    def __init__(self, tag, path):
        ProvenanceObject.__init__(self, tag)
        self._path = path
```

```
        self._json = {}

    def __repr__(self):
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'name': self._tag, 'path':
         ↪ self._path}
        return self._json
```

4.4.5. MethodType

Essa classe é responsável por definir os possíveis tipos de um *Method*, sendo apenas um Enum.

```
from enum import Enum

class MethodType(Enum):
    COMPUTATION = 'COMPUTATION'
    INSTRUMENTATION = 'INSTRUMENTATION'
    EXTRACTION = 'EXTRACTION'
    PROVENANCE = 'PROVENANCE'
```

4.4.6. Performance

Como o nome sugere, essa classe é responsável por encapsular as informações referentes a performance dos objetos da proveniência retrospectiva do *dataflow*. Na classe é necessário indicar a natureza do método (4.4.5) ao qual está sendo avaliada a performance, uma *tag* e o tempo de início da execução.

```
class Performance(ProvenanceObject):
    def __init__(self, tag, method_type, start_time,
     ↪ task_id=None, description=None, invocation=None
                , task_sub_id=None, end_time=None):
        ProvenanceObject.__init__(self, tag)
        self._description = description
        self._method_type = method_type
        self._invocation = invocation
        self._task_id = task_id
        self._task_sub_id = task_sub_id
        self._start_time = start_time
        self._end_time = end_time
        self._json = {}

    def __repr__(self):
```

```
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json = {'description': self._description,
         ↪ 'method': self._method_type,
                      'startTime': self._start_time,
                       ↪ 'endTime': self._end_time}
        return self._json
```

4.4.7. StatusType

Essa classe é responsável por definir os possíveis *status* para uma *Task*, sendo apenas um Enum.

```
from enum import Enum

class StatusType(Enum):
    READY = 'READY'
    RUNNING = 'RUNNING'
    FINISHED = 'FINISHED'
```

4.4.8. Dataflow

Essa classe é a principal classe da proveniência retrospectiva, sendo responsável por encapsular todas as propriedades necessárias para definir uma *Task* (tarefa) segundo o modelo de dados de proveniência utilizado pelo *DfAnalyzer* (PROV-Df). Seus atributos são *tag* (nome da tarefa), dataflow_tag (tag do *dataflow* ao qual pertence), transformation_tag (*tag* da transformação a qual pertence), os demais campos serão apresentados de forma literal no código abaixo:

```
class Task(ProvenanceObject):
    def __init__(self, tag, dataflow_tag,
     ↪ transformation_tag, resource, workspace, status,
     ↪ id, performances=[],
                output="", error="", sub_id="",
                 ↪ invocation="", dependency="",
                 ↪ files=[], idatasets = []):
        ProvenanceObject.__init__(self, tag)
        self._id = id
        self._sub_id = sub_id
        self._dataflow_tag = dataflow_tag
        self._transformation_tag = transformation_tag
        self._resource = resource
        self._workspace = workspace
        self._performances = performances
```

```
        self._invocation = invocation
        self._status = status
        self._output = output
        self._error = error
        self._dependency = dependency
        self._files = files
        self._idatasets = idatasets
        self._json = {}

    def __repr__(self):
        return json.dumps(self._json, indent=2)

    def get_json(self):
        self._json =
        {'tag': self._tag, 'id': self._id,
        'sub_id': self._sub_id, 'resource':
         ↪ self._resource,
        'workspace': self._workspace, 'invocation':
         ↪ self._invocation,
        'status': self._status,
        'output': self._output, 'error': self._error,
         ↪ "dataflow": self._dataflow_tag,
        "transformation": self._transformation_tag,
         ↪ "performances": self._performances,
        "dependency": self._dependency, "files" :
         ↪ self._files, "sets" : self._idatasets}

        return self._json
```

5. Estudo de Caso

Com o intuito de validar a solução proposta foi criado um exemplo utilizando o *PySpark* e o *Jupyter Notebook*. Os códigos desenvolvidos para o estudo de caso e os do DfA-PyWrapper foram disponibilizados em [Repositório a]. O *PySpark* é uma API em *Python* para criação de aplicações *Spark*. O exemplo em questão é um *dataflow* simples composto por duas transformações. Como o intuito era apenas demonstrar a utilização da ferramenta e comparar seu uso para instrumentar código em *Python* utilizando as chamadas ao *Provenance Data Gatherer* desenvolvido em *Java*, foram escolhidas transformações "*dummy*"que seriam basicamente operadores de mapeamento (*map*) que não modificariam os dados. A implementação dessa transformação está apresentada em Nessa seção apresentaremos, primeiramente, a estrutura do experimento já instrumentado utilizando o a API em *Python* (*wrapper*) desenvolvido para o trabalho. Em seguida, para efeitos de comparação, apresentaremos como seria o processo de instrumentação de um código utilizando o PDG (*Provenance Data Gatherer*) em *Java* utilizando chamadas RPCs. A parte comum aos códigos será apresentada nessa seção, deixando apenas as partes específicas com a instrumentação para serem apresentadas nas subseções dos exemplos.

5.1. Exemplo *PySpark*: Configuração

```
import uuid
from subprocess import Popen, PIPE, STDOUT
from datetime import datetime
from random import randint
import findspark
findspark.init('/opt/spark-2.2.0-bin-hadoop2.7')
from pyspark import SparkConf, SparkContext, SQLContext
from pyspark.sql import DataFrameWriter, Row
import sys
sys.path.append('../')
from PDE import PDE

# Configura o spark
conf = ( SparkConf()
         .setMaster("local[*]")
         .setAppName('pyspark')
        )
sc = SparkContext(conf=conf)
sqlContext = SQLContext(sc)
base_url = "http://localhost:22000/"
```

5.2. Exemplo *PySpark*: Transformação (*Map*)

```
class MapTransformation(object):
    def __init__(self,rdd):
        self._rdd = rdd

    def do_nothing(self):
        return self._rdd.map(lambda s: s)
```

5.3. Exemplo *PySpark* : DfA-PyWrapper

Nessa subseção será apresentada o código instrumentado utilizando o *Wrapper* em *Python* desenvolvido como trabalho para a disciplina.

5.3.1. Proveniência Prospectiva

Nessa etapa será apresentado o código referente à utização do *DfA-PyWrapper* para extração de dados da proveniência prospectiva. É importante frisar que, para a proveniência prospectiva, não teríamos muito impacto em relação à utilização da versão em Java. Isso ocorre, pois mesmo que o código do cientista estivesse em *Python*, a proveniência prospectiva poderia ser especificada em um arquivo separado.

```
# Configura proveniência prospectiva.
```

```
# Define o dataflow.
dataflow = PDE.dataflow("spark-example")

# Define as transformações.
dt1 = PDE.transformation("dt1")
dt2 = PDE.transformation("dt2")

# Define os atributos.
att1 = PDE.attribute("att1",
 ↪ PDE.attribute_type().TEXT.value)
att2 = PDE.attribute("att2",
 ↪ PDE.attribute_type().NUMERIC.value)
att3 = PDE.attribute("att3",
 ↪ PDE.attribute_type().TEXT.value)
att4 = PDE.attribute("att4",
 ↪ PDE.attribute_type().NUMERIC.value)
att5 = PDE.attribute("att5",
 ↪ PDE.attribute_type().TEXT.value)
att6 = PDE.attribute("att6",
 ↪ PDE.attribute_type().NUMERIC.value)


# Define os extratores.
ext1 = PDE.extractor("ext1",
 ↪ PDE.extractor_cartridge().EXTRACTION.value,
PDE.extractor_extension().CSV.value)
ext2 = PDE.extractor("ext2",
 ↪ PDE.extractor_cartridge().EXTRACTION.value,
 ↪ PDE.extractor_extension().CSV.value)
ext3 = PDE.extractor("ext3",
 ↪ PDE.extractor_cartridge().EXTRACTION.value,
 ↪ PDE.extractor_extension().CSV.value)

# Adiciona os extratores aos atributos.
att1._extractor = ext1._tag
att2._extractor = ext1._tag
att3._extractor = ext2._tag
att4._extractor = ext2._tag
att5._extractor = ext3._tag
att6._extractor = ext3._tag

# Define os Datasets.
ds1 = PDE.set("ds1", PDE.set_type().OUTPUT.value)
ds2 = PDE.set("ds2", PDE.set_type().INPUT.value)
ds3 = PDE.set("ds3", PDE.set_type().OUTPUT.value)
```

```
# Adiciona os atributos aos datasets.
ds1._attributes = [att1.get_json(),att2.get_json()]
ds2._attributes = [att3.get_json(),att4.get_json()]
ds3._attributes = [att5.get_json(),att6.get_json()]

program  = PDE.program("testando","path")

# Adiciona os extratores aos datasets.
ds1._extractors = [ext1.get_json()]
ds2._extractors = [ext2.get_json()]
ds3._extractors = [ext3.get_json()]

# Adiciona os datasets às transformações.
dt1._sets = [ds1.get_json()]
dt2._sets = [ds2.get_json(),ds3.get_json()]

dt1._programs = [program.get_json()]
# Adiciona as transformações ao dataflow
dataflow._transformations =
 ↪  [dt1.get_json(),dt2.get_json()]

# Realiza post na api restful do dfa para inserir o
 ↪  dataflow.
PDE.ingest_dataflow_json(base_url,dataflow.get_json())
```

5.3.2. Proveniência Retrospectiva

Nessa etapa será apresentado o código referente à utização do *DfA-PyWrapper* para extração de dados da proveniência retrospectiva.

```
# Configura proveniência retrospectiva.
t1 = PDE.task("t1",dataflow._tag, dt1._tag, "resource",
 ↪  "workspace",PDE.status_type().READY.value, "1")
t2 = PDE.task("t2",dataflow._tag, dt2._tag, "resource",
 ↪  "workspace",PDE.status_type().READY.value, "2")

# Cria performances para t1.
pf1 = PDE.performance("pf1",
    PDE.method_type().COMPUTATION.value,"ti" ,
    t1._id)

# Inicializa a task1
t1._status = PDE.status_type().RUNNING.value

# Cria input.
rdd = sc.parallelize([Row(att1=str(uuid.uuid4()),
```

```
    att2=randint(0,100)) for x in range(5)])

m1 = MapTransformation(rdd)

r2 = m1.do_nothing()

# Escreve em csv.
df1 = sqlContext.createDataFrame(r2)

# Arquivo de saída de t1.
f1 = PDE.file("dt1-output.csv", ".")

df1.coalesce(1).write.save(path=f1._tag, format='csv',
 ↪ mode='append', sep=',')

# Adiciona informações à tarefa 1.
pf1._end_time = "tf"
t1._performances.append(pf1.get_json())
t1._files.append(f1.get_json())
t1._status = PDE.status_type().FINISHED.value

# Armazena t1 na base.
PDE.ingest_task_json(base_url,t1.get_json())

# Lê do csv.

# Adiciona arquivo de entrada à t2.
t2._files.append(f1.get_json())
# Criar arquivo de performance para t2.
pf2 = PDE.performance("pf2",
    PDE.method_type().COMPUTATION.value,
    "ti" ,t2._id)

df2 = (sqlContext.read
         .format("com.databricks.spark.csv")
         .option("header", "false")
         .load(f1._tag))

# Carrega informações na task2.
m2 = MapTransformation(df2.rdd)

r3 = m2.do_nothing()

# Adiciona informações à tarefa 2.
pf2._end_time = "tf"
t2._performances.append(pf1.get_json())
```

```
t2._status = PDE.status_type().FINISHED.value

# Armazena t2 na base.
PDE.ingest_task_json(base_url,t2.get_json())
```

5.4. Exemplo *PySpark* : PDG *Java*

5.4.1. Proveniência Prospectiva

Nessa etapa será apresentado o código referente à utização do *PDG (Provenance Data Gatherer)* para extração de dados da proveniência prospectiva.

```
# Configura proveniência prospectiva.

#Define o dataflow
p1 = Popen(['java', '-jar',
 ↪  'PDG-1.0.jar','-dataflow','-tag','spark-example'])
p1.wait()
# Define as transformações.
p2 = Popen(['java', '-jar',
 ↪  'PDG-1.0.jar','-transformation','-dataflow',
       'spark-example','-tag','dt1'])
p2.wait()
p3= Popen(['java', '-jar',
 ↪  'PDG-1.0.jar','-transformation','-dataflow',
       'spark-example','-tag','dt2'])
p3.wait()

# Define os programas
p4 = Popen(['java', '-jar', 'PDG-1.0.jar','-program',
            '-dataflow','spark-example','-transformation',
            'dt1','-name','testando','-filepath','path'])
p4.wait()

# Define os Datasets.
p5 = Popen(['java', '-jar',
 ↪  'PDG-1.0.jar','-set','-dataflow',
       'spark-example','-transformation','dt1','-tag',
       'ds1','-type','output'])
p5.wait()

p6 = Popen(['java', '-jar',
 ↪  'PDG-1.0.jar','-set','-dataflow',
       'spark-example','-transformation','dt2','-tag',
       'ds2','-type','input'])
p6.wait()
```

```
p7 = Popen(['java', '-jar',
 ↪  'PDG-1.0.jar','-set','-dataflow',
       'spark-example','-transformation','dt2','-tag',
       'ds3','-type','output'])
p7.wait()

# Define os extratores.
p8 = Popen(['java', '-jar', 'PDG-1.0.jar','-extractor',
        '-dataflow','spark-example','-transformation',
            'dt1','-set','ds1','-tag','ext1',
            '-algorithm','extraction','csv'])
p8.wait()

p9 = Popen(['java', '-jar', 'PDG-1.0.jar','-extractor',
        '-dataflow','spark-example','-transformation',
        'dt2','-set','ds2','-tag','ext2','-algorithm',
        'extraction','csv'])
p9.wait()

p10 = Popen(['java', '-jar', 'PDG-1.0.jar','-extractor',
        '-dataflow','spark-example','-transformation',
        'dt2','-set','ds3','-tag','ext3','-algorithm',
        'extraction','csv'])
p10.wait()

# Define os atributos.
p11 = Popen(['java', '-jar', 'PDG-1.0.jar','-attribute',
        '-dataflow','spark-example','-transformation',
        'dt1','-set','ds1','-name','att1','-type','text',
        '-extractor','ext1'])
p11.wait()

p12 = Popen(['java', '-jar', 'PDG-1.0.jar','-attribute',
        '-dataflow','spark-example','-transformation',
        'dt1','-set','ds1','-name','att2','-type',
        'numeric','-extractor','ext1'])
p12.wait()

p13 = Popen(['java', '-jar',
 ↪  'PDG-1.0.jar','-attribute','-dataflow',
       'spark-example','-transformation','dt2',
        '-set','ds2','-name','att3','-type',
        'text','-extractor','ext2'])
p13.wait()

p14 = Popen(['java', '-jar', 'PDG-1.0.jar','-attribute',
```

```
        '-dataflow','spark-example','-transformation',
        'dt2','-set','ds2','-name','att4','-type',
        'numeric','-extractor','ext2'])
p14.wait()

p15 = Popen(['java', '-jar', 'PDG-1.0.jar','-attribute',
        '-dataflow','spark-example','-transformation',
        'dt2','-set','ds3','-name','att5','-type','text',
        '-extractor','ext3'])
p15.wait()

p16 = Popen(['java', '-jar', 'PDG-1.0.jar','-attribute',
        '-dataflow','spark-example','-transformation',
        'dt2','-set','ds3','-name','att6','-type',
        'numeric','-extractor','ext3'])
p16.wait()
```

5.4.2. Proveniência Retrospectiva

Nessa etapa será apresentado o código referente à utização do *PDG* para extração de dados da proveniência retrospectiva do exemplo criado em *PySpark*.

```
# Configura proveniência retrospectiva e execução do
 ↪ dataflow.

# Configura Task 1
p17 = Popen(['java', '-jar', 'PDG-1.0.jar','-task',
            '-dataflow','spark-example',
            'transformation','dt1',
             '-id','t1','-subid',
             '1','-resouce','resource',
             '-workspace','workspace',
             '-status','running'])
p17.wait()

# Cria performances para t1.
p18 = Popen(['java', '-jar', 'PDG-1.0.jar','-performance',
             '-dataflow','spark-example',
             '-transformation','dt1','-task','t1',
             '-starttime','-description',
             'pf1','-computation','description'])
p18.wait()

# Cria input.
rdd = sc.parallelize([Row(att1=str(uuid.uuid4()),
        att2=randint(0,100)) for x in range(5)])
```

```
m1 = MapTransformation(rdd)

r2 = m1.do_nothing()

# Escreve em csv.
df1 = sqlContext.createDataFrame(r2)

# Arquivo de saída de t1.
p19 = Popen(['java', '-jar',
 ↪  'PDG-1.0.jar','-file','-dataflow',
             'spark-example','transformation','dt1',
             '-id','t1','-name','dt1-output.csv',
             '-path','.'])
p19.wait()

df1.coalesce(1).write.save(path='dt1-output.csv',
 ↪  format='csv',
                            mode='append', sep=',')

# Adiciona informações à tarefa 1.
p20 = Popen(['java', '-jar', 'PDG-1.0.jar','-performance',
             '-dataflow','spark-example',
             '-transformation','dt1','-task','t1',
             '-endtime','-description','pf1',
             '-computation','description'])
p20.wait()

# Modifica o estado da tarefa 1 para finished.
p21 = Popen(['java', '-jar', 'PDG-1.0.jar','-task',
             '-dataflow','spark-example','transformation',
             'dt1','-id','t1','-resouce','resource',
             '-workspace','workspace','-status',
             'finished'])
p21.wait()
# Lê do csv.

# Configura Task 2.
p22 = Popen(['java', '-jar',
 ↪  'PDG-1.0.jar','-task','-dataflow',
             'spark-example','transformation','dt2',
             '-id','t2','-resouce','resource','-workspace'
             ,'workspace','-status','running'])
p22.wait()

# Adiciona arquivo de entrada à t2.
```

```
p23 = Popen(['java', '-jar', 'PDG-1.0.jar','-file',
             '-dataflow','spark-example',
             'transformation','dt2','-id','t2',
             '-name','dt1-output.csv','-path','.'])
p23.wait()

# Criar arquivo de performance para t2.
p24 = Popen(['java', '-jar', 'PDG-1.0.jar','-performance',
             '-dataflow','spark-example',
             '-transformation','dt2','-task','t2',
             '-starttime','-description',
             'pf2','-computation','description'])
p24.wait()

df2 = (sqlContext.read
         .format("com.databricks.spark.csv")
         .option("header", "false")
         .load('dt1-output.csv'))

# Carrega informações na task2.
m2 = MapTransformation(df2.rdd)

r3 = m2.do_nothing()

# Adiciona informações à tarefa 2.
p25 = Popen(['java', '-jar', 'PDG-1.0.jar','-performance',
             '-dataflow','spark-example',
             '-transformation','dt2','-task','t2',
             '-endtime','-description','pf2',
             '-computation','description'])
p25.wait()

# Modifica o estado da tarefa 2 para finished.
p26 = Popen(['java', '-jar', 'PDG-1.0.jar',
             '-task','-dataflow','spark-example'
             ,'transformation','dt2','-id',
             't2','-resouce','resource','-workspace'
             ,'workspace','-status','finished'])
p26.wait()
```

6. Conclusão e trabalhos futuros

A utilização da abstração de *workflows* científicos e como ferramenta de modelagem de simulações computacionais, por meio da execução em SGWfCs(Sistemas de Gerências de *Workflows* Científicos), tende a reduzir a sobrecarga da gerência por parte dos cientistas. Além disso, com o aumento da capacidade computacional, simulações computacionais intensivas em dados tem se tornado cada vez mais comuns. Nesse cenário, faz-se necessário

a rastreabilidade do dado que pode ser obtida, por exemplo, por meio da utilização de ferramentas que possibilitem captura de dados de proveniência.

Conforme discutido ao longo do relatório, a ferramenta desenvolvida por esse trabalho se baseia na modelagem para dados de proveniência Prov-Df, abordada em 3.2. O objetivo da aplicação é facilitar o processo da instrumentação do código, por parte do cientista, sem que seja necessária a utilização de RPCs (*Remote Procedure Calls*). Apesar da utilização de RPCs prejudicarem o tempo de execução de um código esse trabalho não visou apresentar a vantagem da utilização da ferramenta desenvolvida por meio da análise do desempenho, não tendo gerado métricas que evidenciassem a melhoria nesse aspecto. Além disso, a solução desenvolvida se integra a API Restful do *Dataflow Analyzer* em vez de escrever as representações (*jsons*) em arquivo. Esse comportamento modular (uma das principais características da arquitetura ARMFUL) pode facilitar a extensão dos modelos utilizados para outros domínios. A escolha da linguagem de programação *Python* se justifica ao analisarmos a grande gama de projetos de áreas como *data science* que são desenvolvidos utilizando a linguagem utilizando ferramentas como o *IPython* (*Jupyter Notebook*).

A extensão da API, com o intuito de criar uma aplicação Web utilizando Flask (*Microframework* para criação de aplicações *Web* em *Python*), para possibilitar a especificação da proveniência prospectiva do *dataflow* por meio de uma interface gráfica amigável para o usuário seria um dos possíveis trabalhos a serem desenvolvidos. Além disso, a utilização do *Flask* pode favorecer o processo de modularização da solução que poderia se tornar uma espécie de *tool-box* para o desenvolvimento de soluções de proveniência podendo, futuramente, modificar até mesmo qual a modelagem de proveniência utilizada.

Referências

Costa, F., Sousa, V., de Oliveira, D., Ocaña, K., Ogasawara, E., Dias, J., and Mattoso, M. (2013). Capturing and querying workflow runtime provenance with prov: A practical approach. page 282–289.

F. Codd, E. (1990). *The Relational Model for Database Management, Version 2*.

Mattoso, M., Dias, J., Ocaña, K., Ogasawara, E., Costa, F., Horta, F., Sousa, V., and de Oliveira, D. (2014). Dynamic steering of hpc scientific workflows: A survey. 46.

Ogasawara, E., de Oliveira, D., Valduriez, P., Dias, J., Porto, F., and Mattoso, M. (2011). An algebraic approach for data-centric scientific workflows. 4:1328–1339.

Repositório. Dfa-pywrapper. `https://github.com/cos831-2017/DfA-PyWrapper`.

Repositório. Dfanalyzer-spark-example. `https://github.com/vssousa/dfanalyzer-spark`.

Silva, V., de Oliveira, D., Valduriez, P., and Mattoso, M. (2016). Analyzing related raw data files through dataflows. *Concurrency and Computation: Practice and Experience*, 28(8):2528–2545. CPE-15-0131.R1.

Silva, V., Leite, J., Camata, J. J., de Oliveira, D., Coutinho, A. L., Valduriez, P., and Mattoso, M. (2017). Raw data queries during data-intensive parallel workflow execution. *Future Generation Computer Systems*, 75(Supplement C):402 – 422.