Um Framework para facilitar o desenvolvimento de aplicativos para extração e processamento de informações de várias fontes

Otávio Calaça Xavier
Instituto Federal de Goiás
Goiânia, Brasil
otavio.xavier@ifg.edu.br

Sandrerley Ramos Pires
Universidade Federal de Goiás
Goiânia, Brasil
sandrerley@ufg.br

Thyago Carvalho Marques
Universidade Federal de Goiás
Goiânia, Brasil
thyago@ufg.br

Eduardo Augusto Santos Garcia
Universidade Federal de Goiás
Goiânia, Brasil
edusantosgarcia@gmail.com

Felipe Pires Saraiva
Universidade Federal de Goiás
Goiânia, Brasil
felipepsaraiva@discente.ufg.br

Anderson Soares da Silva
Universidade Federal de Goiás
Goiânia, Brasil
anderson@inf.ufg.br

***Abstract*—The information extraction process from several sources aiming to generate a Big Data environment is a complex task. The variables involved such as the volume of information, the speed which new information appears, and the variety of their origins characterize this complex environment. This situation leads developers to deal with a set of details that the available technology requires for their efficient use. The result is a data extraction process little productive. Also, the currently available tools are designed to cover predefined scenarios and are hard to customize. This work proposes a framework to support the developer to deal with this task programmatically. The purpose of the framework is to provide an easy way to develop a distributed resilient pipeline of tasks. It abstracts details of the database manipulation and message queuing manipulation. By using a message queuing system, it can distribute the load between multiple nodes. The framework gives to the developer the possibility to create cohesive and well-coupled modules through the queues, allowing the scalability of the data extraction structure. Thus, the developer can focus on understanding the source data structures he aims to extract. The developer does not need to worry with issues such as queue management, load balancing in a distributed environment, control of database connections and even the writing of boring SQL commands, without adding complexity to the development. Compared to the currently available tools, the proposed framework is lightweight, easier to use and designed for developers, to be used programmatically instead of using the drag-n-drop approach, resulting in significant productivity gains.**

***Resumo ou Resumen*—O processo de extração de informações de diversas fontes com o objetivo de gerar um ambiente de Big Data é uma tarefa complexa. As variáveis envolvidas, como o volume de informações, a velocidade com que novas informações aparecem e a variedade de suas origens caracterizam esse ambiente complexo. Essa situação leva os desenvolvedores a lidar com um conjunto de detalhes que a tecnologia disponível requer para seu uso eficiente. O resultado é um processo de extração de dados pouco produtivo. Além disso, as ferramentas disponíveis atualmente são projetadas para cobrir cenários específicos e são difíceis de serem adaptadas. Este trabalho propõe um framework para apoiar o desenvolvedor a lidar com esta tarefa de forma produtiva. O objetivo da estrutura é fornecer uma maneira fácil de desenvolver um pipeline de tarefas resilientes e distribuídas. Ele abstrai detalhes da manipulação do banco de dados e da manipulação do enfileiramento de mensagens. Usando um sistema de enfileiramento de mensagens, ele pode distribuir a carga entre vários nós. O Framework dá ao desenvolvedor a possibilidade de criar módulos coesos e bem acoplados através das filas, permitindo a escalabilidade da estrutura de extração de dados. Assim, o desenvolvedor pode se concentrar em entender as estruturas de dados de origem que pretende extrair. Ele não precisa se preocupar com questões como gerenciamento de filas, balanceamento de carga em um ambiente distribuído, controle de conexões de banco de dados e até mesmo a escrita de enfadonhos comandos SQL, reduzindo a complexidade ao desenvolvimento. Em comparação com as ferramentas disponíveis atualmente, a estrutura proposta é leve, mais fácil de usar e projetada para desenvolvedores. Ela foi concebida para ser usada de forma programática em vez de usar a abordagem arrastar e soltar, resultando em ganhos de produtividade significativos.**

***Palavras-chave*—Framework, Message Broker, Extração de informação, ETL.**

I. Introdução

O processo de extração de informações para um ambiente de Big Data [1] envolve três aspectos principais que o torna uma tarefa complexa para o desenvolvedor, sendo eles o volume de informação a ser tratado, a velocidade com que novas informações aparecem e a diversidade de origens dessas informações. As aplicações desenvolvidas são normalmente complexas, consumidoras de recursos de máquina e utilizam diversas tecnologias de suporte ao seu funcionamento [2].

Ao desenvolvedor, ou engenheiro de dados, é importante que ele invista a maior parte de seu tempo em identificar

locais onde buscar informações, entender o formato com elas estão armazenadas na origem, definir os mecanismos de busca dessas informações e, finalmente, disponibilizá-las de maneira acessível para análise e consumo. A complexidade na implementação dos mecanismos de ingestão pode levar a perda de produtividade no desenvolvimento desse tipo de aplicação.

Assim, no momento do desenvolvimento, dadas as diversas complexidades com que o desenvolvedor tem que lidar, é importante que ele conte com um framework que o apoie na tarefa de implementação das rotinas de extração de informações. Nesse framework, tarefas usuais como tratamento de filas, modularização visando o uso de processamento distribuído e a manipulação de banco de dados devem se tornar o mais transparente possível para o desenvolvedor.

Um aspecto importante para a construção de aplicações de extração de dados é o desacoplamento das atividades a serem realizadas no processo. As atividades no processo de extração (requisição, transformação, validação, carga, entre outras) podem possuir diferentes níveis de complexidade e necessitar de diferentes quantidades de tempo para serem executadas. Ao serem desacopladas, é possível utilizar vários agentes em atividades mais lentas a fim de otimizar o tempo do processo como um todo.

O desacoplamento das atividades pode ser realizado com a utilização de protocolos de mensageria ao invés de chamadas diretas a funções ou métodos. Assim, ao final da execução de uma atividade, seu resultado é enfileirado para posterior consumo pela atividade subsequente no processo. Um dos protocolos mais utilizados é o AMQP [3]. Essa abordagem permite modularizar as aplicações de extração e estabelecer um modelo seguro de comunicação entre elas, permitindo o processamento distribuído e paralelo dos módulos extratores.

Existem disponíveis diversos softwares no mercado, podendo se citar o *RabbitMQ* [4], o *ActiveMQ* [5] e o *MSQM* [6]. Esses produtos oferecem implementações do protocolo AMQP e podem ser instalados como serviços gerenciadores de mensagens (ou *message brokers*). Por se tratar de produtos para uso geral, exigem significativo domínio e conhecimento por parte do desenvolvedor em relação ao protocolo e ao servidor em si.

Para agilizar o processo de desenvolvimento, é importante que o framework ofereça a capacidade de tratamento de mensagens e de uso do banco de dados, sem, contudo, ter que tratar da complexidade inerente aos software de mensagens, bem como na escrita de comandos de manipulação de banco de dados, que muitas vezes tomam tempo do desenvolvedor dado o aspecto burocrático de tais comandos. Acrescenta-se o fato de que o framework deve oferecer a possibilidade de controlar automaticamente o processo de execução das atividades, aumentando a quantidade de agentes a fim de consumir mais rapidamente as filas de mensagens, caso seja necessário.

Este trabalho apresenta um framework para o desenvolvimento de aplicações de extração de dados que permite ao desenvolvedor focar no processo de extração e modelagem do armazenamento, dando-lhe um ambiente de fácil uso para o desenvolvimento de suas aplicações. O artigo está estruturado em uma seção 2 com uma fundamentação teórica para facilitar o processo de leitura do artigo e uma visão de produtos correlatos, uma seção 3 que apresenta os requisitos definidos para o framework, bem como a estrutura da solução apresentada, a seção 4 mostra um exemplo de uso do framework e, finalmente, a seção 5 as conclusões deste trabalho.

II. Gerenciador de Mensagens e Trabalhos Correlatos

Esta seção apresenta uma visão geral do que é um sistema de gerenciamento de mensagem que é a base fundamental para o framework aqui proposto. Em seguida é mostrado um panorama de produtos correlatos.

A. Gerenciador de Filas

O processo de extração de informação na web, além das características já descritas anteriormente, possui alguns modelos padronizados de ação. Dois modelos típicos de extração de informação podem ser feitos da seguinte forma:

a) Baixa de um arquivo estruturado de algum site na internet, leitura e seleção dos dados desse arquivo e gravação dos resultados em banco de dados. Por exemplo, um arquivo CSV[1] contendo a lista de empresas de uma determinada localidade.

b) Requisições individualizadas utilizando formulário de consulta. Nesse modelo, a identificação única da informação é dada e objetiva-se a obtenção dos demais dados do indivíduo. Por exemplo, uma relação de CNPJ em que se deseja buscar a informação de se existe algum processo em andamento contra essas empresas. Nesse caso, a entrada são os argumentos de busca, e as atividades são: efetuar a busca de forma individual de cada um dos argumentos, selecionar as informações coletadas e gravá-las em banco de dados.

É possível notar que a velocidade de cada uma das subatividades do processo de extração de informação pode ser significativamente diferente entre elas. Assim, se uma aplicação no molde tradicional for utilizada, haverá passos do processo que serão gargalos, prejudicando a performance dos outros passos.

O uso de um sistema gerenciador de filas pode evitar os gargalos, pois ele reduz o acoplamento entre as subatividades, fazendo com que as atividades mais rápidas não necessitem esperar a execução daquelas que são mais lentas. A comunicação entre as atividades é realizada através de filas. Assim, caso uma atividade seja muito mais rápida que a subsequente, as mensagens ficarão acumuladas na fila de comunicação entre elas. Com isso, é possível executar múltiplos agentes para a execução da atividade mais lenta, reduzindo o acúmulo e aumentando a performance do processo como um todo.

A Figura 1 mostra um esquema simplificado de funcionamento de um gerenciador de filas. Nota-se que um gerenciador de mensagens pode ter muitos clientes se comunicando com ele simultaneamente. A quantidade de filas que podem ser criadas é também grande, sendo isso limitado pelo recurso de máquina. Um cliente pode estar enviando mensagens para alguma fila, outros podem estar retirando mensagens das filas e até mesmo, fazer as duas coisas em uma mesma tarefa.

[1] CSV: é a sigla para *Comma Separated Values*, ou Valores Separados por Vírgula, um formato muito utilizado para representação de dados tabulares.

Diversas vantagens podem ser obtidas com esse tipo de software. Duas grandes vantagens são:

a) A possibilidade de maximizar a performance de uma tarefa que envolve muitas filas, delegando maior número de clientes para efetuar o tratamento de mensagens;

b) Distribuir os processos de tratamento de mensagens em um ambiente de processamento distribuído, elevando significativamente a capacidade total de processamento.

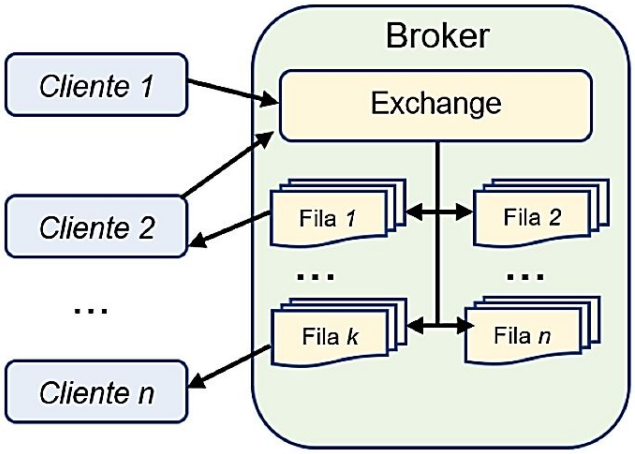

Fig. 1: Esquema simplificado de funcionamento de um gerenciador

B. Trabalhos Relacionados

O processo de extrair dados de várias fontes distintas para armazená-los em um banco de dados é comumente conhecido como ETL (*Extract-Transform-Load*). Esse é um processo muito utilizado na indústria e para construir armazéns de dados (*data warehouses*). Assim, existem diversas ferramentas comerciais e de código aberto disponíveis voltadas à essa demanda.

Microsoft, IBM e Oracle são algumas das empresas que possuem soluções comerciais que dominam esse mercado. Ainda, algumas soluções de código aberto também são amplamente usadas, como o Talend [7] e o Pentaho [8]. A Gartner publica anualmente os Quadrantes Mágicos para Ferramentas de Integração de Dados [9] onde é possível ver as mais utilizadas.

Nessas ferramentas a principal forma de utilização é através de uma interface gráfica em que o fluxo dos dados é representado e construído de forma totalmente visual. Essa abordagem deixa as soluções mais amigáveis para quem não programa, mas limita bastante a flexibilidade das operações.

Outras soluções adotam a abordagem de construir o pipeline através de código, como o petl [10], que é simples e apresenta várias formas de extrair e salvar dados, mas não possibilita recursos mais avançados como paralelismo ou execução em várias máquinas. O ETLator [11] e o SimpleETL [12] são ferramentas mais completas inspiradas no, mais antigo, pygrametl [13]. Elas são fáceis de usar, com pouca configuração, e possuem muitos recursos avançados como processamento em lotes, paralelismo, cache dos dados lidos, entre outros. Entretanto as duas não permitem a distribuição da execução em várias máquinas ou a pausa do processamento, e são voltadas para a construção de data warehouses em que os dados serão modelados em esquema estrela, deixando a modelagem do pipeline de ETL atrelada a esse esquema e restringindo a abrangência da solução.

III. A Abordagem Proposta

Esta seção apresenta os requisitos definidos para que o framework possa realmente apoiar o processo de implementação de aplicações extratoras de informação. Após os requisitos, é mostrada a solução encontrada para que o desenvolvedor tenha maior produtividade no desenvolvimento da aplicação.

A. Os Requisitos do Framework

Para o desenvolvimento de camadas de software que facilitem o processo de desenvolvimento de aplicações de extração de informação, os requisitos foram separados em duas camadas distintas, a camada de utilização de filas e a camada de banco de dados. A Figura 2 ilustra como essas duas camadas de software se comunicam com a aplicação do usuário.

O projeto de desenvolvimento do framework considerou algumas restrições tecnológicas para sua construção. O Sistema Gerenciador de Banco de Dados (SGBD) utilizado é o PostgreSQL [14], o sistema de gerenciador de mensagens é o RabbitMQ e a linguagem utilizada para o desenvolvimento das camadas e aplicações é o Python [15]. Apesar dessa restrição tecnológica, o gerenciador de mensagens pode ser substituído por qualquer outro que suporte o protocolo AMQP, bem como o banco de dados, sem a necessidade de modificação no código fonte, apenas com atualizações de configuração.

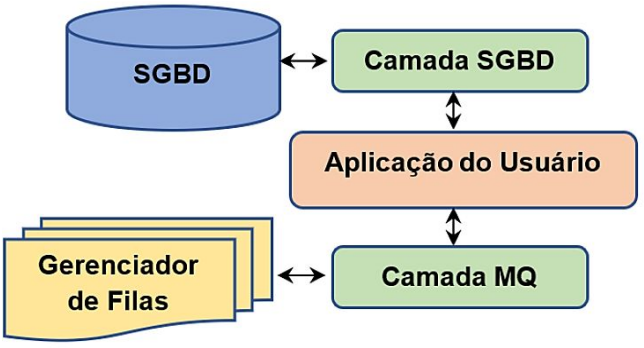

Fig. 2: Relação de camadas que compõem uma aplicação com o framework.

B. Manipulação de dados em SGBD

O primeiro requisito planejado foi para facilitar o processo de conexão ao sistema gerenciador de banco de dados, bem como o controle automático das conexões. Um arquivo de configuração, único para todo o ambiente de desenvolvimento, contém os parâmetros necessários para a realização das conexões ao SGBD. Para o desenvolvedor basta um import na classe gerenciadora e a inicialização de um objeto que será o SGBD. A seguir é apresentado um exemplo de como se codificar o objeto de acesso ao SGBD.

```
from tasks.data import DataSourceSQL
oBD = DataSourceSQL()
```

Após a conexão, o objeto *oBD* oferece todos os serviços clássicos de uma aplicação que usa banco de dados, sendo eles: *save(), update(), delete(), query(), beginTransaction(), endTransaction()*. A semântica dos nomes dos métodos já descreve como se dá a sua utilização.

O método *query()* executa um comando SELECT no SGBD, tendo dois parâmetros: uma *string* com o comando em si e uma lista de campos que irão compor as condições do comando. A substituição dos valores dos campos no comando é feita pelo próprio método. Isso evita falhas de segurança como SQL Injection [16]. O retorno dos dados será um generator do Python, ou seja, uma lista de dados que poderá ser consumida sob demanda.

Dado o grau de flexibilidade que envolve a construção de um comando SELECT, optou-se por recebê-lo inteiramente como parâmetro. O trecho de código a seguir ilustra como utilizar o método *query()*. Nele é realizada uma consulta com tendo como argumento o número 3, que será utilizado em substituição ao *placeholder %s* no comando SQL.

```
sql = "SELECT *
FROM empresa.fatura_a_interpretar
WHERE ftr_status > %s"
result = oBD.query(sql, [16])
```

Os métodos *beginTransaction()* e *endTransaction()* estabelecem uma transação para garantir a atomicidade de um bloco de ações que serão realizadas no banco de dados. No framework, esses dois métodos têm uma função a mais, que é o controle de conexões entre tarefas paralelas. Se uma aplicação faz uma única conexão com o banco de dados e ativa tarefas paralelas, o controle das transações de banco de dados fica inviável. Assim, ao chamar o método *beginTransaction()*, é retornado um objeto para tratar exclusivamente aquela transação que se inicia, a qual é encerrada com o método *endTransaction()*.

O modo de uso de tais métodos é relativamente similar aos já disponíveis em diversos outros frameworks. Ao invocar o método beginTransaction(), o usuário recebe o objeto oBD para o acesso ao SGBD, dando-lhe uma conexão exclusiva para rodar sua transação. Após o término da transação, o comando endTransaction() faz o COMMIT e a liberação da conexão. O trecho abaixo ilustra o uso conjunto desses serviços.

```
oBD_t = oBD.beginTransaction()
... atividades da transação
oBD.endTransaction(oBD_t)
```

O desenvolvedor pode não utilizar os comandos de controle de transação, mas deve ter em mente que se sua implementação for utilizar paralelismos entre tarefas, ele pode ter problemas de quebra de integridade no banco de dados.

Para facilitar a utilização dos demais serviço, padronizou-se que os parâmetros a serem passados a eles, no que se refere aos campos de uma tabela, devem estar no formato de um campo dicionário (*dict*) do Python, onde os campos chaves (*keys*) devem conter os nomes dos campos da tabela que serão envolvidos no comando e os campos valores (*values*) devem conter os valores a serem utilizados no comando SQL.

Os três comandos de manipulação *save(), update()* e *delete()* devem passar como primeiro parâmetro o nome da tabela que sofrerá a atuação do comando. Os comandos *update()* e *delete()* têm como segundo parâmetro o campo condição para que se possa indicar o critério (da cláusula WHERE) que particularizará o escopo de atuação do comando. A não informação desse parâmetro incorrerá na atuação em toda tabela. No caso dos comandos *save()* e update(), deve-se passar a relação dos campos que participarão do comando, no caso do *update()*, os campos que serão modificados e, no caso do *save()*, os campos que serão inseridos.

Para o comando *save()*, se for passado uma lista de campos dicionários, isto será interpretado como inserção de mais que uma *tupla*. O framework gerará um único comando INSERT para inserção de toda lista de *tuplas* informadas. Esse recurso tende a reduzir muito o tempo de execução de uma tarefa de extração. Todos os campos dicionários da lista devem ter a mesma estrutura.

Finalmente, para os métodos, *save(), update()* e *delete()*, se o método *beginTransaction()* não tiver sido chamado anteriormente, um *COMMIT* será executado após o comando em questão. Para todos esses métodos pode-se informar na lista de parâmetros as opções *returning=field_name e on_conflict=action*. Essas opções permitem tratamentos adicionais aos comandos para retorno de algum valor ou ações no caso de ocorrência de conflitos na execução do comando.

Segue um trecho de código completo em que é feito o uso dos três serviços de manipulação citados nesta subseção.

```
reg_a_incluir = {'rg':1744209,
'nome':'Nome hipotético',
'acesso':100}
reg_a_alterar = {'acesso': 100}
oBD_t = oBD.beginTransaction()
# Inclusão do registro
oBD_t.save('usuario', reg_a_incluir)
# alteração do nível de acesso do usuário
condicao = 'rg = 6992453'
oBD_t.update('usuario', condicao, reg_a_alterar)
# exclusão de um usuário
condicao = 'rg = 7886539'
oBD_t.delete('usuario', condicao)
pg.endTransaction(oBD_t)
```

Observe que com a interface oferecida pelo framework, apenas o comando SELECT deverá ser codificado na aplicação, os demais serão automaticamente montados pela classe de manipulação do SGBD, facilitando significativamente o trabalho de codificação.

C. Integração de módulos através de filas

O tratamento de filas oferece diversos recursos para aplicações que manipulam grande quantidade de dados. Entre esses pode-se citar: a distribuição do tratamento de filas em várias máquinas; a compartimentalização das aplicações, tornando-as mais manuteníveis; e a utilização de mais tarefas paralelas para ajudar a consumir uma fila mais rapidamente.

O objetivo com o framework de tratamento de fila é que cada desenvolvedor tenha em mente que ao desenvolver uma aplicação de extração de informações, ela deve ser pensada como um conjunto de tarefas encadeadas formando um pipeline.

A Figura 3 mostra um exemplo de definição de um pipeline para uma aplicação de extração de informação. Nela, é possível observar que se trata de um pipeline em três etapas. A primeira faz busca de informações em formato bruto em diferentes origens, banco de dados de sistemas

legados, sites na web ou até mesmo em documentos não formatados. O volume de informações tratados é normalmente grande e a tarefa gera sua saída para uma fila, chamada

de dados brutos, permitindo que uma outra tarefa (ou outras) possa tratar o conteúdo dessa fila. Na sequência, a segunda etapa do pipeline pode ser invocada, visto que já existem elementos na fila para serem consumidos. Nessa etapa os dados brutos são tratados e formatados para serem inseridos no banco de dados da aplicação. A saída já tratada é colocada em uma segunda fila, chamada de dados a inserir. Finalmente, quando houver dados nessa última fila, o gerenciador chamará a tarefa relativa à terceira etapa do pipeline, sendo que nesse exemplo, o objetivo é a inserção no banco de dados da aplicação.

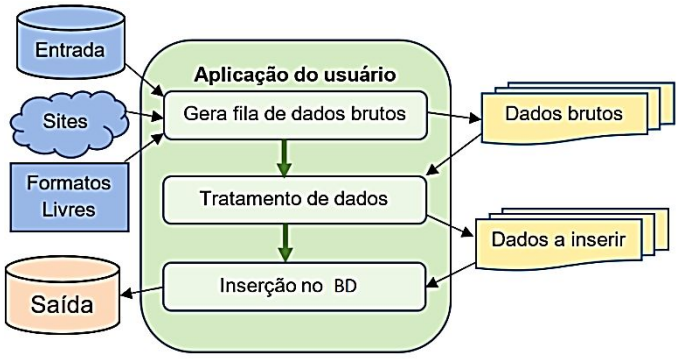

Fig. 3: Exemplo de um pipeline com três etapas.

A fim de facilitar o processo de implementação de um pipeline utilizando filas, definiu-se, neste trabalho, os seguintes requisitos para o uso de mensagens e filas:

a) Inicialização: para criação de um pipeline, o primeiro elemento é responsável por iniciar o processo de extração. Trata-se de uma função ou método que retorna uma lista de valores a serem enfileirados na primeira fila do pipeline. Opcionalmente, no caso de filas muito longas, o retorno pode ser na forma de um *generator*, através do comando *yield* do Python, garantindo maior paralelismo ao processo. Assim, essa primeira função já começa a enfileirar os valores antes mesmo que toda a lista tenha sido montada, permitindo que o próximo elemento do pipeline já possa começar a consumir as mensagens.

b) Elementos Intermediários: os demais elementos do pipeline consomem as mensagens das filas de saída dos elementos anteriores, processam e retornam novas mensagens em suas filas de saída. O processo de geração de saída para uma nova fila é feito da mesma forma que descrito para o primeiro elemento do pipeline.

c) Finalização: o último elemento do pipeline que normalmente executa uma operação no banco de dados, não gera dados para uma outra fila. Nesse caso basta encerrar a atividade sem retornar nenhum dado ao gerenciador.

d) Tolerância a Falhas: caso qualquer erro ocorra durante o processamento de uma mensagem, ela é enfileirada, junto com os detalhes da exceção lançada, em uma fila de falhas específica do elemento em questão do pipeline. Essa mensagem de erro pode então ser consumida por um outro serviço ou ser analisada pelo desenvolvedor do pipeline.

Para atender os requisitos propostos, foi desenvolvido o módulo Python denominado *pipeline_ampq* que é capaz de oferecer todos os recursos descritos nesta seção. O módulo consiste em duas classes principais, *PythonAmqpTask* e *PythonAmqpPipeline* e outras duas classes derivadas, *ageablePythonAmqpTask* e *BatchPythonAmqpTask*, descritas abaixo.

a) *PythonAmqpPipeline*: classe que tem a responsabilidade de interagir com as aplicações do cliente. Ela é a única classe que o usuário do framework irá utilizar. Em seu construtor, ela recebe uma lista de *tuplas* contendo o nome da tarefa, função a ser executada e outros parâmetros opcionais. Com isso, ela abstrai completamente a criação e manipulação das mensagens e filas.

b) *PythonAmqpTask*: classe derivada de *threading.Thread*. Ela é responsável por definir uma atividade ou elemento do pipeline e é utilizada apenas internamente na classe *PythonAmqpPipeline*. Seu construtor recebe como parâmetros o nome da tarefa, a fila de entrada, a fila de saída, a função a ser executada e um parâmetro opcional com configurações específicas da tarefa. Com isso, ela cria um consumidor na fila de entrada que irá consumir as mensagens e passar como parâmetro para a função informada. Ao final da execução da função, seu retorno é enfileirado na fila de saída. Em caso de erro ou exceção, é enfileirada uma mensagem contendo a exceção e o conteúdo da mensagem de entrada em uma fila de falhas.

c) *PageablePythonAmqpTask*: classe derivada de *PythonAmqp-Task*. Ela adiciona a característica de paginação ao elemento do pipeline. Assim, ao invés de retornar uma lista como todos os elementos, serão utilizadas listas parciais com tamanho pré-definido. Ela recebe os parâmetros adicionais para tamanho de página (*page_size*) e deslocamento (*offset*) para construção das páginas. Sua utilização é comum no primeiro elemento do pipeline, quando não é possível a utilização de *generator* e a entrada possui muitos elementos.

d) *BatchPythonAmqpTask*: classe derivada de *PythonAmqpTask*. Ela possui um recurso adicional para tratamento de mensagens em lote. Ela recebe os parâmetros adicionais *batch_size* e *batch_timeout*. Com isso, ao invés de consumir a mensagem na fila de entrada e já a passar para a função informada, ela armazena as mensagens em uma fila interna até que o tamanho do lote (*batch_size*) ou o tempo limite (*batch_timeout*, dado em segundos) sejam atingidos. Quando um dos dois é atingido, ela chama a função informada passando a lista com todas as mensagens acumuladas. Sua utilização é muito comum no último elemento do pipeline para evitar inserções individuais de registros, aumentando a performance de execução do pipeline.

O trecho de código mostrado a seguir ilustra um exemplo de declaração de um pipeline. Nessa declaração serão utilizadas todas as possibilidades de parâmetros disponibilizados pelo framework.

```
from pipeline_amqp import PythonAmqpPipeline
```

```
pipeline = PythonAmqpPipeline([
   ('bf_1_download', donwload_bf, {'type': 'pageable',
    'page_size': 1000}),
   ('bf_2_formata_saida', gera_saida_bf, {'workers': 4,
    'failure_processor': save_failure}),
   ('bf_3_salvar', salva_bf, {'type': 'batch',
    'batch_size': 200, 'batch_timeout': 10}
])
```

O trecho mostra que foi declarado um pipeline com três passos/elementos. O controlador criará duas filas para que haja a conexão entre os elementos. Essas filas serão chamadas de bf_1_download_output e bf_2_formata_saida_output. No mesmo módulo Python que se declarou o pipeline, deve-se implementar as três funções que realizarão as tarefas de extração. No código mostrado, as funções são *download_bf(), gera_saida_bf()* e *salva_bf()*.

Essas duas últimas funções serão chamadas sempre que houver mensagens nas filas de saída dos elementos predecessores.

Além das filas de conexão entre os elementos do pipeline, outras três filas serão criadas para tratamento de erros: *bf_1_download_failures, bf_2_formata_saida_failures* e *bf_3_salvar_failures.*

Na declaração da fila pode-se observar que parâmetros adicionais foram informados. Segue uma descrição de cada um deles:

a) 'type': 'pageable': parâmetro responsável por informar que a classe *PageablePythonAmqpTask* deve ser utilizada.

b) '*page_size': 1000*: esse parâmetro é utilizado pela classe *PageablePythonAmqpTask* para configurar o tamanho da página. Assim, as mensagens serão enfileiradas em páginas de 1000 elementos na fila de saída em questão.

c) 'workers': 4: essa opção indica quantas tarefas (*threads*) serão geradas para efetuar o tratamento paralelo da fila. Quando um elemento do pipeline tem velocidade significativamente maior que os outros, essa opção pode ajudar a melhorar a performance total do pipeline.

d) 'type': 'batch': indica que a classe *BatchPythonAmqpTask* deve ser utilizada. Com isso, é possível, por exemplo, a inserção de blocos de registros, aumentando a eficiência do processo.

e) '*batch_size': 200*: no caso de recebimento de dados em lote, essa opção define o tamanho do bloco. No caso, a tarefa consumidora será chamada recebendo 200 elementos da fila de cada vez. Quando a fila estiver com menos de 200 elementos, será passado o número de mensagens que nela estiver.

f) 'batch_timeout': 10: no caso de recebimento de dados em lote, essa opção define o tempo limite de espera para construção de um lote. No caso, o tempo limite será 10 segundos. Ou seja, mesmo que o lote não tenha atingido o tamanho configurado na opção anterior, ele será enviado para processamento após o tempo limite.

g) 'failure_processor': save_failure: este parâmetro indica que se ocorrer alguma exceção durante o tratamento de uma mensagem específica, o framework irá chamar a rotina de nome informado nos parâmetros, no caso *save_failure*, para fazer um tratamento mais adequado da exceção. A rotina cujo nome foi informado nesta opção de tratamento de falha deve estar implementada dentro do código feito pelo desenvolvedor.

Cabe ao desenvolvedor fazer a importação do framework em seu programa, instanciar a classe *PythonAmqpPipeline* como foi mostrado no trecho de código de definição do pipeline.

IV. Um Exemplo de Utilização do Framework

A fim de ilustrar como se desenvolve uma aplicação utilizando os recursos do framework aqui proposto, este trabalho apresenta um exemplo de como se criar a aplicação de extração de informações. Embora esta apresentação seja feita a partir de trechos de código, todos eles pertencem a um mesmo arquivo Python.

O exemplo ilustra um processo de extração de informações de beneficiários do Bolsa Família. Mensalmente, o portal de Dados Abertos do Brasil libera para download uma planilha no formato CSV e compactada no formato ZIP para agilizar o processo de download. Nela contém os dados de pagamento de aproximadamente treze milhões de beneficiários do programa. O objetivo é capturar essa planilha e atualizar os dados das tabelas de beneficiário e de pagamento existentes em uma aplicação de *Bigdata*.

A aplicação a ser mostrada tem três passos básicos, sendo eles:

a) O primeiro é o download da planilha compactada, após isso é feito a descompactação e a inserção dos registros da tabela CSV em uma fila para serem processados;

b) O segundo faz a análise dos registros de pagamento e gera saídas de dados para o beneficiário, caso não esteja cadastrado e dos seus dados do pagamento; e,

c) Finalmente, o terceiro passo é fazer a gravação das informações nas tabelas de beneficiário e de pagamento.

Um módulo denominado *bolsafamilia.py* faz as atividades particulares de análise da planilha e de formatação dos dicionários com os dados de beneficiário e de pagamento. Como esse módulo não é fundamental para entender como se utiliza o framework, ele não será detalhado aqui neste artigo. O módulo main.py é o código que será detalhado no exemplo. Ele cria o pipeline e implementa os seus serviços básicos. Assim, os passos importantes para o entendimento de como se constrói a aplicação são os descritos a seguir:

A. Definição do Pipeline

O trecho de código abaixo mostra como ficou a definição do pipeline, bem como a instanciação da classe principal do framework.

```
pipeline = PythonAmqpPipeline([
('bf_1_download_csv', download_csv),
('bf_2_formatar_dados', formatar_dados,
{'workers': 2}),
('bf_3_salva_dados', salvar_dados,
{'type': 'batch',
'batch_size': 200})
])
```

Pode-se observar que são utilizadas duas filas, a primeira com os dados brutos dos pagamentos do Bolsa Família e a

segunda com as informações já tratadas e prontas para serem inseridas em banco de dados.

O desenvolvedor fez a opção por manter executando duas tarefas paralelas do processo de análise dos dados de pagamento. Esse valor pode ser ajustado com o uso, aumentando ou diminuindo o número de tarefas.

Observe no código que a última etapa do pipeline irá receber blocos de 200 mensagens para que possa executar a inserção em bloco no banco de dados, aumentando a performance da aplicação.

Segue uma apresentação dos códigos dos serviços *donwload_csv, formatar_dados* e *salvar_dados*.

1) *O serviço de download da tabela de pagamento.* A implementação deste serviço é apresentada a seguir.

```
def download_csv():
    try:
      for pagamento in
        bolsafamilia.busca_pagtos(args.ano, args.mes)
        yield pagamento
    except ValueError:
          raise Exception('Problema na leitura de
          pagamentos' + args.ano + ' args.mes)
```

Pode-se observar no código que um serviço do módulo *bolsafamilia* é chamado e ele faz todo o processo de busca na internet da planilha, faz a descompactação e retorna a planilha na forma de uma lista para o módulo chamador. O módulo irá retornar os pagamentos para a fila através do comando *yield*, garantindo que não será necessário tratar os treze milhões de pagamentos para começar as atividades do próximo passo do pipeline.

Pode-se observar também que existe o tratamento de exceções. Caso algum problema ocorra durante a execução a exceção será gerada e as mensagens serão passadas ao framework. Ele por sua vez irá gravar o erro em uma fila de erros para serem tratados posteriormente, além de gerar a saída no log de execução da aplicação como WARN.

2) *O serviço de análise e formatação dos dados de pagamento.* A implementação deste serviço é apresentada a seguir.

```
def formatar_dados(linha_pagto):
    try:
        pagamento, beneficiario =
            bolsafamilia.gera_dict_bf(linha_pagto)
        return {'pagamento': pagamento,
            'beneficiario': beneficiario}
    except ValueError:
        raise Exception ('Erro na interpretação')
```

O Código apresentado faz a análise do pagamento e gera dois tipos diferentes de dicionários para serem gravados. O registro de cadastro do beneficiário e os dados de seu primeiro pagamento para gravação. O gerenciador do framework irá criar duas tarefas em paralelo para tratar os dados de pagamentos. À medida que esses dados são processados eles são encaminhados para a fila de saída que irá assim, dar início ao processo de gravação.

A ocorrência de um erro durante o tratamento de um elemento na fila não indica que os outros elementos sofrerão com isso, parando a execução geral do processo. Na ocorrência de um erro é gravado em uma fila de falhas os dados necessários para futura análise e correção do problema. Essa característica resiliente do framework é uma de suas maiores vantagens.

3) *O serviço de gravação dos dados.* A implementação deste serviço é apresentada a seguir.

```
def salvar_dados(lista_pagtos):
    try:
        pgt = pg.BeginTransaction()
        pg.save('bolsafamilia.pagamento',
            lista_pagtos[:]['pagamento']):
        pg.savel('bolsafamilia.beneficiario',
            lista_pagtos[:]['beneficiario'])
        pg.EndTransaction(pgt)
    except ValueError:
        raise Exception('problemas na inserção na tabela')
```

V. Resultados Obtidos com o uso em Produção

A disponibilização do Framework permitiu a implementação de aplicações e uma percepção de seu uso. As próximas subseções detalham e analisam esses resultados.

A. *Aplicações desenvolvidas com o Framework.*

O framework está sendo amplamente utilizado em um projeto de pesquisa que visa avaliar o ranqueamento de empresas com relação a diversos critérios como capacidade de serem sonegadores e capacidade de quitarem seus débitos tributários. No total foram implementadas quatorze rotinas de extração de dados utilizando este pipeline, permitindo a clara percepção de sua utilidade para o processo de implementação.

As análises apresentadas na próxima seção estão sustentadas pela avaliação dessas quatorze rotinas em produção em um intervalo de seis meses.

B. *Ganho de produtividade ao usar o framework.*

O desenvolvimento do framework provocou um ganho significativo de produtividade em uma equipe de desenvolvedores em que ele foi submetido ao uso. Pode-se citar os seguintes aspectos em que a sua utilização passou essa percepção de ganho de produtividade:

a) A quantidade de linhas de código foi reduzida, em média, a menos da metade que no modelo de implementação utilizado antes do uso do framework. Essa percepção foi possível com a conversão de várias aplicações de extração de informações realizadas para o teste do framework;

b) A transparência dada ao tratamento de mensagem faz com que o usuário possa desenvolver suas aplicações utilizando todo o potencial desse recurso sem ter que conhecer com profundidade a sua forma de utilização. Sem a necessidade de treinamentos adicionais. Esse é um ganho direto percebido com o uso do framework;

c) O controle automático das conexões, dispensando o desenvolvedor de ter que se preocupar com isso, gerou facilidade de implementação e melhor administração dos recursos de utilização do SGBD;

d) A redução de comandos SQL no código. Embora os comandos não sejam complexos, eles são burocráticos e trabalhosos de serem escrito;

e) O processo de manutenção das aplicações devido a um maior nível de abstração dado a elas foi sensivelmente facilitado. A utilização do framework estabelece um padrão para as aplicações, reforçando o ganho de manutenibilidade.

VI. Conclusão

O trabalho se propôs a desenvolver um framework para construção de pipelines de extração de dados com facilidade para tratamento de mensagens e de acesso ao banco de dados. O objetivo com esse framework é a produção de códigos mais enxutos, estáveis sem deixar de ter todas as vantagens de tarefas paralelas que utilizem massivamente esses dois recursos citados.

Acredita-se que este produto detalhado e exemplificado neste trabalho atinge plenamente os seus objetivos iniciais. A utilização dos serviços de tratamento de mensagens ficou totalmente transparente ao desenvolvedor e o tratamento de banco de dados significativamente facilitado. Acredita-se que a solução dada para o tratamento de transações paralelas seja simples e fácil de ser assimilada, embora não totalmente transparente ao usuário.

A equipe de desenvolvedores de aplicações de extração de informação que utilizou o framework tem a clara percepção de que há ganho de produtividade na produção de software.

1) Trabalhos Futuros

Uma vez que o framework está desenvolvido e sendo utilizado diariamente por uma equipe de desenvolvedores, engenheiros e cientistas de dados, é possível destacar os seguintes aprimoramentos que estão e serão feitos ao projeto:

a) Execução de testes utilizando o método científico para mensuração do ganho de produtividade que o framework promove para a equipe que o utiliza.

b) Adição da possibilidade de escalabilidade automática. Ou seja, que o framework tenha recursos para que sejam identificados acúmulos em filas automaticamente e inicie, caso haja recurso disponível, novos agentes para processamento das mensagens.

c) Suporte a outros protocolos de mensageria como Stomp[2] e MQTT[3] visando a utilização em outros casos de uso, como ingestão de dados para a internet das coisas.

d) Desenvolvimento de interface Web para monitoramento das mensagens e execução dos pipelines. Mesmo que a maioria dos gerenciadores de mensagens já possuam interfaces para monitoramento das filas, uma interface própria poderia reduzir a complexidade do monitoramento e adicionar elementos específicos, como o monitoramento dos pipelines inteiros ao invés de apenas filas isoladas.

Agradecimentos

Agradecimentos à FAPEG - Fundação de Amparo à Pesquisa do Estado de Goiás e ao IFG - Instituto Federal de Educação, Ciência e Tecnologia por financiarem o projeto.

[2] Site Oficial do Protocolo Stomp: https://stomp.github.io/

[3] MQTT é a sigla para MQ Telemetry Transport, um protocolo muito utilizado na internet das coisas para envio de dados de sensores.

Referências

[1] F. N. R. Machado. *"Big Data: O futuro dos dados e aplicações*". Érica, São Paulo, SP, Brazil, 2018.

[2] U. Sivarajah, M. M. Kamal, Z. Irani, V. Weerakkody. *"Critical analysis of Big Data challenges and analytical methods"*. Journal of Business Research 70 (2017), 263–286, 2017.

[3] S. Vinoski. *"Advanced Message Queuing Protocol"*. Internet Computing, IEEE 10 (12 2006), 87-89, 2006. https://doi.org/10.1109/MIC.2006.116

[4] Rabbit MQ 2020. Open Source Message Broker. https://www.rabbitmq.com/.

[5] Active MQ 2020. Open Source Protocol Developed by Apache. https://activemq.apache.org/.

[6] Msqm 2020. Microsoft Message Queuing. https://docs.microsoft.com/enus/previous-versions/windows/desktop/legacy/ms711472(v=vs.85).

[7] Talend 2020. Talend Data Fabric™. https://www.talend.com.

[8] Pentaho 2020. Pentaho Platfom. https://www.hitachivantara.com/enus/products/data-management-analytics/pentaho-platform.htm.

[9] Quadrante 2020. Quadrantes Mágicos do Gartner de 2019 para Ferramentas de integração de dados e iPaaS Empresarial. https://www.informatica.com/br/dataintegration-magic-quadrant.html.

[10] Petl 2020. petl - Extract, Transform and Load (Tables of Data). https://petl.readthedocs.io.

[11] M. Radonić, I. Mekterović. *"ETLator - a scripting ETL framework"*. In 40th International Convention on Information e Communication Technology, Electronics and Microelectronics (MIPRO). Opatija, Croácia, 1349–1354, 2017. https://doi.org/10.23919/MIPRO.2017.7973632

[12] C. T. Andersen, Ove and Torp Kristian. "*SimpleETL: ETL Processing by Simple Specifications"*. In 20th International Workshop on Design, Optimization, Languages and Analytical Processing of Big Data. Vienna, Austria, 1–6, 2019.

[13] S. K. Jensen, T. B. Pedersen, C. Thomsen, O. Andersen. *"Business Intelligence and Big Data. Programmatic ETL"*, Vol. 324. Springer, Cham, Chapter 2, 21–50, 2018. https://doi.org/10.1007/978-3-319-96655-7

[14] PostgreSQL 2020. PostgreSQL. https://www.postgresql. org.

[15] Python 2020. Python Programming Language. https://www.python.org.

[16] W. G. Halfond, J. Viegas, A. Orso. *"A classification of SQL-injection attacks and countermeasures"*. In Proceedings of the IEEE international symposium on secure software engineering, Vol. 1. IEEE, 13–15, 2006.