GABRIEL MARQUES DE MELO

PROPOSTA DE CONSTRUÇÃO DE UM DATA LAKEHOUSE DE CÓDIGO ABERTO

LAVRAS – MG

2021

GABRIEL MARQUES DE MELO

PROPOSTA DE CONSTRUÇÃO DE UM DATA LAKEHOUSE DE CÓDIGO ABERTO

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Graduação em Ciência da Computação para a obtenção do título de Bacharel.

Prof. DSc. Erick Maziero
Orientador

LAVRAS – MG

2021

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos

da Biblioteca Universitária da UFLA

de Melo, Gabriel Marques

Proposta de construção de um Data Lakehouse de código aberto / . 1^{a} ed. rev., atual. e ampl. – Lavras : UFLA, 2021.

79 p. : il.

Monografia (TCC)–Universidade Federal de Lavras, 2021.

Orientador: Prof. DSc. Erick Maziero.

Bibliografia.

1. TCC. 2. Monografia. 3. Dissertação. 4. Tese. 5. Trabalho Científico – Normas. I. Universidade Federal de Lavras. II. Título.

CDD-808.066

GABRIEL MARQUES DE MELO

PROPOSTA DE CONSTRUÇÃO DE UM DATA LAKEHOUSE DE CÓDIGO ABERTO

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Graduação em Ciência da Computação para a obtenção do título de Bacharel.

APROVADA em 19 de novembro de 2021.

Prof. DSc. Erick Maziero
Orientador

LAVRAS – MG
2021

Dedico aos meus pais, Elisângela Marques Carvalho e Valério Lopes de Melo, meu irmão Leonardo Marques de Melo e minha noiva Giulya Carvalho de Almeida.

AGRADECIMENTOS

Agradeço aos meus pais, irmão e familiares por todo o esforço e dedicação investidos em minha educação.

Agradeço à minha noiva que sempre esteve ao meu lado durante o meu percurso acadêmico.

Sou grato pela confiança e apoio do professor Erick Maziero, orientador do meu trabalho e responsável por fazer deste processo um período de aprendizado constante.

Agradeço, também, à Universidade Federal de Lavras e todo o seu corpo docente, por estes anos de crescimento e exposição a um ambiente tão plural e rico em conhecimento.

The saddest aspect of life right now is that science gathers knowledge faster than society gathers wisdom.
(Isaac Asimov)

RESUMO

A transformação digital tem levado cada vez mais empresas para um contexto de processamento e armazenamento em nuvem (*cloud computing*). Contudo, para projetos de menores dimensões ou de propósitos didáticos, o custo desses serviços, principalmente os relacionados ao processamento de dados, pode ser maior que o orçamento disponível, ou representar um elevado gasto em projetos que não possuem um retorno financeiro direto. Além disso, pode haver o receio em se armazenar e processar dados sigilosos e pessoais em servidores na nuvem, por motivos de segurança e privacidade. Existem, entretanto, alternativas para esses processamentos em nuvem, como dispositivos de processamento de baixo custo e, até mesmo, computadores pessoais. Ademais, uma abordagem que vem ganhando espaço no mercado nos últimos anos é o conceito de *Data Lakehouse*, que, em resumo, traz a flexibilidade e baixo custo de armazenamento dos consolidados *Data Lakes* aliados à consistência das transações ACID dos *Data Warehouses*. Este trabalho propõe uma arquitetura de *Data Lakehouse* de código aberto, *on premise*, e de baixo custo de infraestrutura, para desenvolvimento de projetos de dados com objetivos didáticos ou de iniciativas de escopo reduzido de dados.

Palavras-chave: *Data Lakehouse*. Código Aberto. Engenharia de Dados.

LISTA DE FIGURAS

Figura 2.1 – Arquitetura Docker comparada a Arquitetura de máquina virtual 14
Figura 2.2 – Instanciando um contêiner a partir de um contêiner: (1) Usuário envia uma requisição, via Docker CLI na máquina hospedeira, para o *dockerd* instanciar um contêiner; (2) o contêiner é criado; (3) A aplicação envia uma requisição, via Docker CLI no contêiner, para o *dockerd* instanciar um novo contêiner; (4) É instanciado um novo contêiner, que é "irmão"do que o solicitou. 16
Figura 2.3 – Representação simplificada do uso do Redis como um *broker* para enfileiramento de tarefas: (1) Nó cliente sinaliza ao broker, através de mensagem, que uma tarefa "x" deve ser executada; (2) Nó executor, que permanece escutando o broker aguardando por mensagens, lê a tarefa "x" e a executa. 17
Figura 2.4 – Exemplo de uma aplicação que utiliza Celery: (1) Programas clientes, escritos em Python, enviam, via API do Celery, requisições contendo a identificação de tarefas (e.g., uma função em Python) e a periodicidade em que devem ser executadas; (2) A API serializa a requisição do usuário e envia a mensagem para o *broker*; (3) Os nós executores, que podem estar em máquina distribuídas, recebem as mensagens, verificam se alguma tarefa pode ser executada, de acordo com o agendamento, e decidem qual nó executará cada uma dessas tarefas. 18
Figura 2.5 – Arquitetura base do Apache Airflow 19
Figura 2.6 – Interface Web do Apache Airflow 21

Figura 2.7 – Captura de tela da interface do Mongo Express que exibe a listagem dos documentos de uma coleção 22
Figura 2.8 – Captura de tela da interface Web do Airbyte que exibe a listagem das conexões existentes 23
Figura 3.1 – Arquitetura CoreDB 30
Figura 3.2 – Interface da aplicação Web do Kylo 33
Figura 3.3 – Arquitetura do ecossistema Smart grid Big data 36
Figura 4.1 – Arquitetura proposta 37
Figura 4.2 – Camada de fontes de dados 38
Figura 4.3 – Camada de ingestão de dados 39
Figura 4.4 – Camada de armazenamento de dados 42
Figura 4.5 – Camada de processamento de dados 43
Figura 4.6 – Camada de análise e visualização de dados 44
Figura 4.7 – Camada de orquestração de tarefas e contêineres 45

LISTA DE TABELAS

Tabela 2.1 – Comparativo entre componentes de dados de bancos de dados relacionais e do MongoDB 21
Tabela 4.1 – Aplicações provisionadas na máquina mestre 47
Tabela 4.2 – Requisitos mínimos recomendados para máquina trabalhadora 48
Tabela 4.3 – Aplicações provisionadas na máquina trabalhadora 48
Tabela 5.1 – Resultados obtidos no teste de carga 52
Tabela 5.2 – Especificações da máquina mestre 53
Tabela 5.3 – Especificações da máquina trabalhadora 53

SUMÁRIO

1 **INTRODUÇÃO** 12
2 **REFERENCIAL TEÓRICO** 14
2.1 **Docker** 14
2.2 **Redis** 17
2.3 **Celery** 17
2.4 **Apache Airflow** 18
2.5 **MongoDB** 20
2.6 **Airbyte** 22
2.7 **MinIO** 25
2.8 **Apache Spark** 25
2.9 **Delta Lake** 26
2.10 **Dremio** 27
2.11 **Apache Superset** 27
2.12 **Gatling** 28
3 **TRABALHOS RELACIONADOS** 29
3.1 ***International network performance and security testing based on distributed abyss storage cluster and draft of data lake framework*** 29
3.2 **CoreDB:** ***A Data Lake Service*** 30
3.3 ***Hardware and software data processing system for research and scientific purposes based on Raspberry Pi 3 microcomputer*** . . . 31
3.4 ***Kylo Data Lakes Configuration deployed in Public Cloud environments in Single Node Mode*** 33
3.5 ***Data Lake Lambda Architecture for Smart Grids Big Data Analytics*** 34
4 **METODOLOGIA** 37
4.1 **Arquitetura proposta** 37

4.2 Infraestrutura do projeto 46
4.3 Fora do escopo da solução 48
4.4 Conjuntos de dados utilizados 49
4.5 Código-fonte 50
4.6 Instanciação da solução 51
5 RESULTADOS 52
5.1 Testes de carga 52
5.2 Custos 52
5.3 Avaliação qualitativa 53
6 CONCLUSÃO 55
REFERÊNCIAS 57
APENDICE A – Exemplo de Código fonte - Dockerfile 60
APENDICE B – Exemplo de Código fonte - docker-compose.yml 61
APENDICE C – Modelo de Dockerfile para aplicação raspadora de dados - Dockerfile 62
APENDICE D – Modelo para arquivo de definição de raspador de dados - crawler_template.py 63
APENDICE E – Modelo para arquivo de definição de fonte de dados - sourceMongoDB.json 64
APENDICE F – Modelo para arquivo de definição de destino de dados - destinationMinIO.json 65
APENDICE G – Modelo para arquivo de definição de conexão do Airbyte - connectorTemplate.json 66
APENDICE H – Modelo para arquivo de definição de DAG do Airflow - 01_modelo-ingestao.py 68
APENDICE I – Modelo para arquivo de definição de script Pyspark - 02_modelo-pyspark.json 71

APENDICE J – Modelo para arquivo de definição de DAG do Airflow - 02_modelo-transformacao.py 74
APENDICE K – Modelo de script para criação de conectores no Apache Superset - create_connections_superset.py 77
APENDICE L – Classe principal da biblioteca cliente da API REST do Portainer 78

1 INTRODUÇÃO

A onda de transformação digital e desenvolvimento de uma mentalidade baseada em dados têm levado cada vez mais corporações para um contexto de processamento e armazenamento em nuvem (*cloud computing*). Segundo Flexera (2021), 54% das empresas têm ao menos uma parcela de seus dados na nuvem pública.

Contudo, para projetos de menores dimensões ou de propósitos didáticos, o custo desses serviços, principalmente os relacionados ao processamento de dados, pode ser maior que o orçamento disponível, ou representar um elevado gasto em projetos que não possuem um retorno financeiro direto, inviabilizando novos potenciais projetos e a entrada e formação de novos profissionais na área. O relatório de McKinsey (2019), mostra que apenas 8% das empresas do Brasil executam eficientemente práticas de capacitação e retenção de talentos para um ambiente digital e analítico.

Além disso, pode haver o receio em se armazenar e processar dados em servidores em nuvem pública, por motivos de segurança e privacidade, uma vez que casos de vazamentos e exposições de dados são cada vez mais frequentes no Brasil e no mundo (BISSO et al., 2020).

Existem, entretanto, alternativas para esses processamentos em nuvem, como dispositivos de processamento de baixo custo (e.g. microcomputadores) e, até mesmo, computadores pessoais.

O armazenamento e organização de grandes massas de dados em provedores de nuvem geralmente é realizado por dois modelos consolidados: *Data Warehouse* e *Data Lake*. O primeiro é implementado em banco de dados relacionais e com foco em desempenho de consultas e análises de dados (ORACLE, 2021). O último é desenvolvido em recursos de armazenamento de objetos (e.g., AWS S3 e Azure Blob Storage) e com suporte a dados estruturados e não estruturados(AMAZON, 2021). Contudo, um novo modelo que vem ganhando espaço no

mercado nos últimos anos é o de *Data Lakehouse* que, em resumo, traz a flexibilidade e baixo custo de armazenamento dos *Data Lakes* aliados à consistência das transações ACID (atomicidade-consistência-isolamento-durabilidade, do inglês, *atomicity-consistency-isolation-durability*) e modelagem dimensional dos *Data Warehouses* (ARMBRUST et al., 2020).

Este trabalho tem como objeto propor uma arquitetura de *Data Lakehouse* baseada em ferramentas de código aberto de amplo mercado, *on premise*, e de baixo custo de infraestrutura, para desenvolvimento de projetos de dados com objetivos didáticos ou de iniciativas de escopo reduzido de dados, onde o tamanho máximo, por cada execução de carga de dados, não supere 1 (um) *gigabyte*. Espera-se que essa plataforma possa servir como ferramenta de suporte para iniciativas educacionais (e.g., disciplinas com enfoque em engenharia de dados).

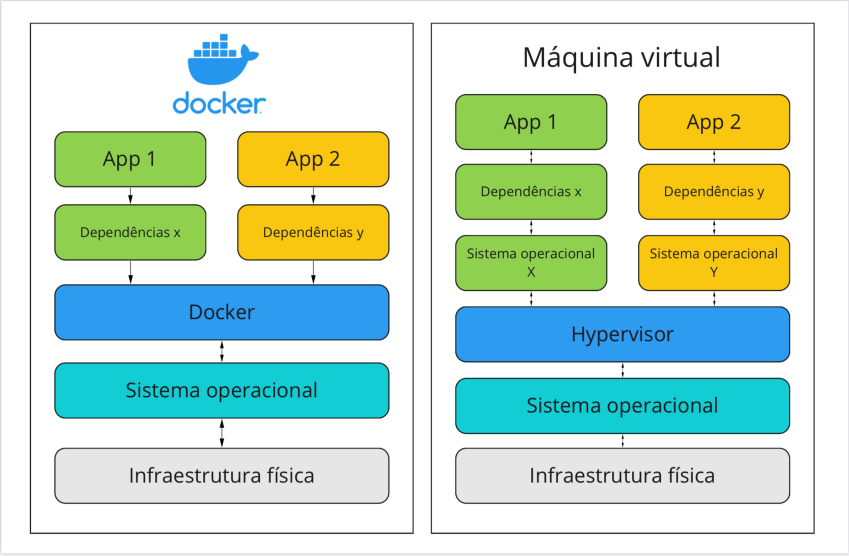
O restante do presente documento é organizado na seguinte estrutura: o Capítulo 2 elucida conceitos e ferramentas relevantes para a compreensão da arquitetura proposta; no Capítulo 3 são levantados trabalhos que possuem similaridade com algum aspecto deste documento e são explicitados os pontos de divergência entre eles; o Capítulo 4 é reservado para a metodologia do desenvolvimento do trabalho, onde são apresentadas a arquitetura, infraestrutura, e a delimitação do escopo do projeto; os resultados obtidos dos testes quantitativos e das avaliações qualitativas sobre a plataforma são apresentados no Capítulo 5; por fim, o Capítulo 6 traz as conclusões acerca dos objetivos alcançados, dos trabalhos futuros e do aprendizado adquirido durante o desenvolvimento do trabalho de conclusão de curso.

2 REFERENCIAL TEÓRICO

2.1 Docker

Docker é uma plataforma de código aberto para execução de aplicações de maneira distribuída, isolada e facilitada (DOCKER, 2021a). Através do Docker, todas as dependências e configurações iniciais de uma aplicação são encapsuladas em uma estrutura conhecida como imagem Docker. Cada imagem em execução é conhecida como um contêiner. Diferentemente do que acontece, por exemplo, em soluções que utilizam máquinas virtuais para execução de aplicações, os contêineres executam em uma camada isolada, sobre o topo do sistema operacional hospedeiro, como ilustrado na Figura 2.1.

Figura 2.1 – Arquitetura Docker comparada a Arquitetura de máquina virtual



Fonte: Imagem do Autor

O Docker possui dois componentes principais: o Docker *daemon* (*dockerd*), responsável por escutar requisições e gerenciar todos os recursos Docker, como imagens, contêineres, volumes e redes; e o Docker CLI, programa utilitário de linha de comando utilizado como interface entre o usuário e o *dockerd*.

2.1.1 Dockerfile

Cada imagem é definida, via código, em um arquivo denominado "Dockerfile". Nesse arquivo, são descritas as operações sequenciais para compor a imagem esperada, incluindo manipulação do sistema de arquivos, compartilhamento de recursos com o sistema operacional hospedeiro, utilização de outra imagem como base, entre outras operações (DOCKER, 2021b). As imagens podem ser armazenadas localmente ou em um repositório remoto (DOCKER, 2021c). Um exemplo de definição de uma imagem Docker, de cunho demonstrativo, pode ser observado no Apêndice A.

2.1.2 Docker Compose

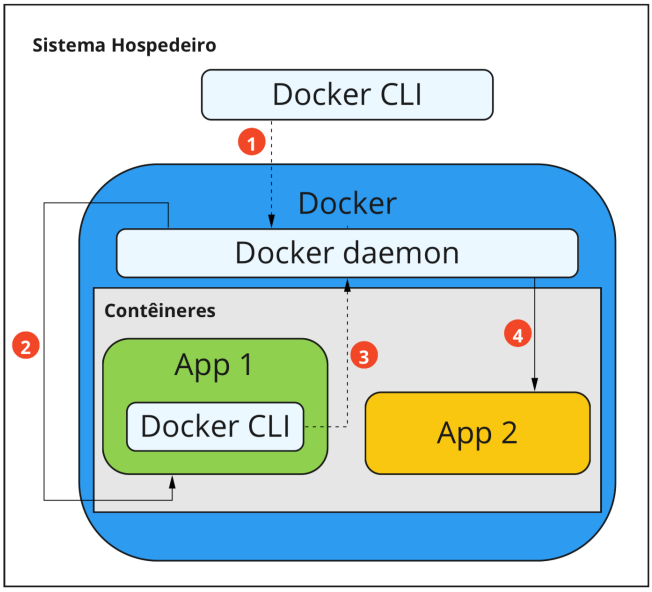
Soluções tecnológicas de mercado modernas, no geral, consistem na composição de diversas aplicações independentes que comunicam entre si. Mesmo em arquiteturas monolíticas, existem serviços distintos para o sistema principal e o banco de dados. No caso de arquiteturas baseadas em microsserviços (PONCE; MÁRQUEZ; ASTUDILLO, 2019), a quantidade de aplicações é consideravelmente maior, o que significa, em um contexto Docker, um número notável de contêineres distintos.

O Docker Compose é uma ferramenta simples para orquestração de instanciações de contêineres. Através de um arquivo, denominado *"docker-compose.yml"*, são definidas as imagens a serem executadas, suas configurações e as dependências entre elas. Dessa forma, é possível instanciar diversos contêineres com um único comando: DOCKER-COMPOSE UP. Um arquivo de exemplo é definido no Apêndice B.

2.1.3 Instanciação de contêineres a partir de um contêiner

O *dockerd*, que executa na maquina hospedeira, pode ser compartilhado com um contêiner que possua o *Docker CLI* instalado, tornando possível instanciar novos contêineres a partir de um contêiner, como ilustrado na Figura 2.2.

Figura 2.2 – Instanciando um contêiner a partir de um contêiner: (1) Usuário envia uma requisição, via Docker CLI na máquina hospedeira, para o *dockerd* instanciar um contêiner; (2) o contêiner é criado; (3) A aplicação envia uma requisição, via Docker CLI no contêiner, para o *dockerd* instanciar um novo contêiner; (4) É instanciado um novo contêiner, que é "irmão"do que o solicitou.



Fonte: Imagem do Autor

2.1.4 Portainer

Portainer (PORTAINER, 2021) é uma plataforma gerenciadora de recursos Docker e Kubernetes [1]. Através de uma interface Web e uma API REST, usuários autenticados são capazes de gerenciar recursos como contêineres, imagens, volumes e redes de um ou mais Docker Daemon conectados, via TCP, bem como monitorar execuções e uso de *hardware* ou se conectar em outras instâncias

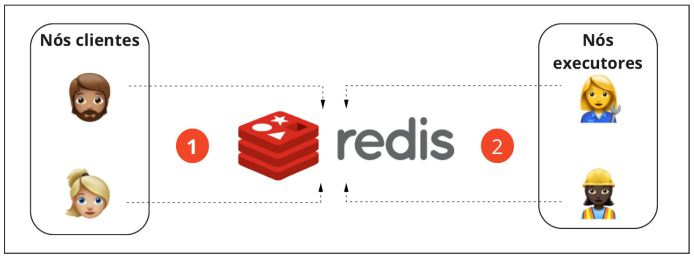
[1] https://kubernetes.io/

Docker. Graças a API REST, é possível programatizar rotinas de início e término de execuções de contêineres em diferentes hospedeiros de forma centralizada, por exemplo.

2.2 Redis

O Redis é um sistema gerenciador de dados transientes, de código aberto, podendo ser usado como um banco de dados, um sistema de cache ou intermediador (do inglês, *broker*) de mensagens (REDIS, 2021). Uma de suas principais aplicações é como um corretor de mensagens para sistemas de enfileiramento e agendamento de tarefas, agindo como um intermediário entre unidades clientes e executoras, como ilustrado na Figura 2.3.

Figura 2.3 – Representação simplificada do uso do Redis como um *broker* para enfileiramento de tarefas: (1) Nó cliente sinaliza ao broker, através de mensagem, que uma tarefa "x" deve ser executada; (2) Nó executor, que permanece escutando o broker aguardando por mensagens, lê a tarefa "x" e a executa.



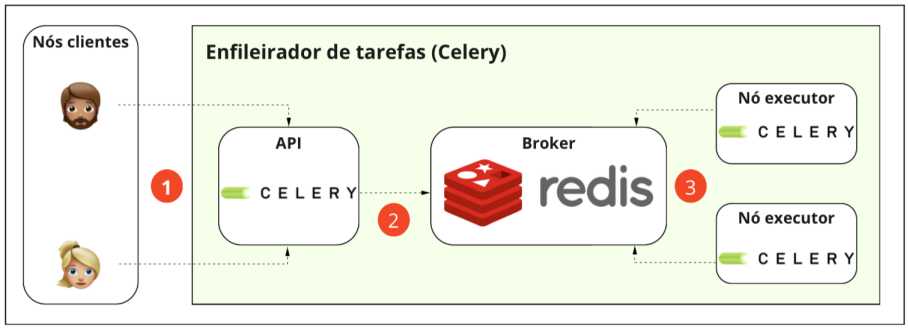
Fonte: Imagem do Autor

2.3 Celery

Celery é um sistema de processamento distribuído, escrito em Python e de código aberto, para enfileiramento e agendamento de tarefas assíncronas (CELERY, 2021). A arquitetura implementada pelo Celery é composta por: uma API para escrita simplificada de aplicações distribuídas utilizando Python; um *broker*, com suporte para os principais bancos de dados transientes, como Redis e Rab-

bitMQ; e um protocolo para execução de tarefas por nós executores. Além disso, fornece uma interface Web para monitoramento e estatísticas de nós executores, bem como opções de escalabilidade, desligamento e reinicialização dos mesmos, através de outro projeto denominado Flower (FLOWER, 2021). Um exemplo simplificado de uma aplicação do Celery é ilustrado na Figura 2.4.

Figura 2.4 – Exemplo de uma aplicação que utiliza Celery: (1) Programas clientes, escritos em Python, enviam, via API do Celery, requisições contendo a identificação de tarefas (e.g., uma função em Python) e a periodicidade em que devem ser executadas; (2) A API serializa a requisição do usuário e envia a mensagem para o *broker*; (3) Os nós executores, que podem estar em máquina distribuídas, recebem as mensagens, verificam se alguma tarefa pode ser executada, de acordo com o agendamento, e decidem qual nó executará cada uma dessas tarefas.



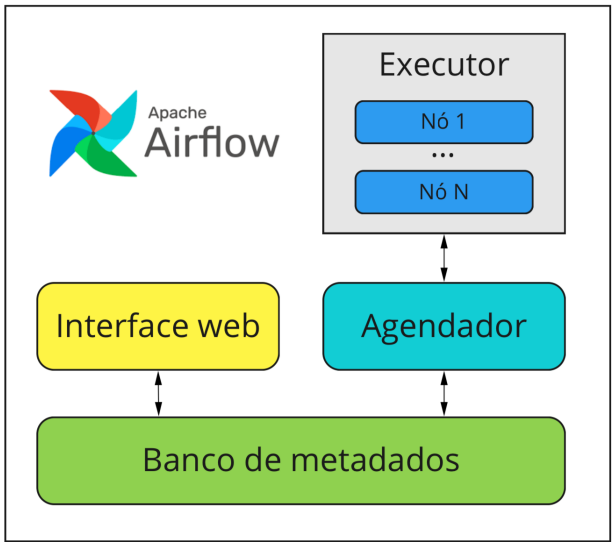
Fonte: Imagem do Autor

2.4 Apache Airflow

Desenvolvido pela empresa Airbnb, em 2014, o Apache Airflow é um orquestrador de fluxo de trabalho escrito em Python e de código aberto (AIRFLOW, 2021). Os *pipelines* desenvolvidos com a ferramenta são organizados como DAGs (grafos acíclicos direcionados, do inglês *directed acyclic graphs*), podendo ser gerados de forma programática, uma vez que consistem em blocos de códigos-fonte, escritos em Python. O Airflow oferece um conjunto extenso de integrações e *plugins* que estendem as funcionalidades da plataforma. Contudo, a arquitetura

interna básica é composta por: um agendador (em inglês, *scheduler*), um executor, um banco de metadados e uma interface Web, como ilustrado na Figura 2.5.

Figura 2.5 – Arquitetura base do Apache Airflow



Fonte: Imagem do Autor

2.4.1 Agendador (*scheduler*)

É o componente responsável pelo disparo das execuções dos pipelines agendados, bem como a atualização dos estados das execuções no banco de metadados. É uma aplicação *multithread* que pode também ser replicada para mitigar sobrecargas ou SPOF (ponto único de falha, do inglês *single point of failure*).

2.4.2 Executor

Executores consistem em classes Python que definem de que maneira (i.e. em qual arquitetura, sob protocolo) os *pipelines* agendados são executados. Para contextos de desenvolvimento ou estudo, pode ser definida como a unidade local da máquina que executa o Airflow de forma sequencial ou paralela, através de *threads* e processos.

Para ambientes produtivos e de maior escala, pode ser configurado para executar de forma distribuída, é o caso, por exemplo, do executor denominado *CeleryExecutor*. Nesse executor, os nós clientes, ilustrados na Figura 2.4, seriam tarefas de uma DAG do Airflow, enquanto os nós executores seriam máquinas remotas.

É ainda possível desenvolver classes customizadas para representar executores com comportamento personalizado (e.g., executor que processa paralelamente apenas em determinadas condições de entrada).

2.4.3 Banco de metadados

É o banco de dados que armazena os metadados de execuções, incluindo as definições das DAGs, agendamentos e *logs* de execução. Em termos de tecnologias suportadas, é possível utilizar PostgreSQL, MySQL ou SQLite como sistema gerenciador de banco de dados (SGBD) através de conectores implementados pelo Apache Airflow.

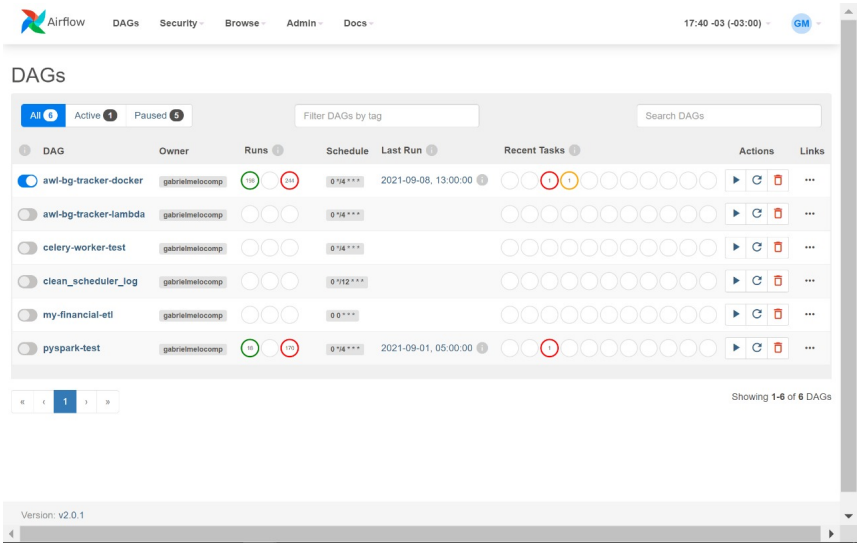
2.4.4 Interface Web

É uma aplicação Web, escrita em Python, que serve de interface ao usuário e onde é possível disparar, monitorar e deletar DAGs, bem como configurar aspectos de segurança e acesso, como ilustrado na Figura 2.6.

2.5 MongoDB

MongoDB é um sistema gerenciador de banco de dados não relacional (NoSQL) de código aberto e baseado em documentos (MONGODB, 2021). Os dados são armazenados em arquivos de formato JSON, que possuem estrutura compatível com as utilizadas por aplicações baseadas em programação orientada a objetos. Em comparação com os bancos de dados relacionais tradicionais, abordagens NoSQL apresentam consideráveis vantagens como: tempo de desenvol-

Figura 2.6 – Interface Web do Apache Airflow



Fonte: Imagem do Autor

vimento, pois esquemas dos dados não precisam ser previamente definidos e se adéquam aos novos dados; escalabilidade, pois a arquitetura que armazena e processa os dados é menos onerosa; e, consequentemente, preço.

Os componentes que compõem a estruturação dos dados no MongoDB também se difere da habitual existente nos banco de dados tabulares e é representada na Tabela 2.1.

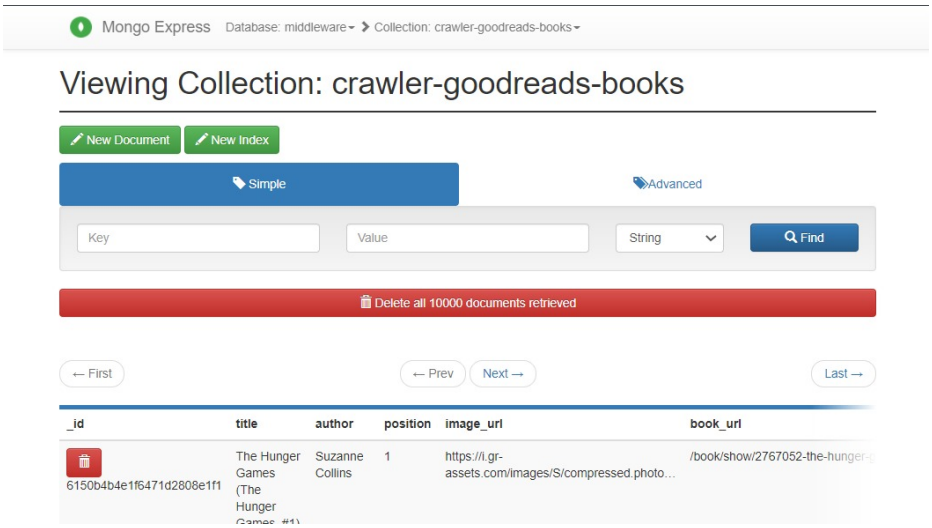
Tabela 2.1 – Comparativo entre componentes de dados de bancos de dados relacionais e do MongoDB

SQL	**MongoDB**
Banco de dados	Banco de dados
Tabela	Coleção
Linha	Documento
Índice	Índice

2.5.1 Mongo Express

Mongo Express é uma interface Web para acesso a instâncias de MongoDB. É possível administrar os recursos dos bancos de dados, bem como consultar, inserir, atualizar e remover de documentos e coleções. A Figura 2.7 representa uma tela de exemplo da interface, onde é possível ver a listagem de documentos de uma coleção.

Figura 2.7 – Captura de tela da interface do Mongo Express que exibe a listagem dos documentos de uma coleção



Fonte: Imagem do Autor

2.6 Airbyte

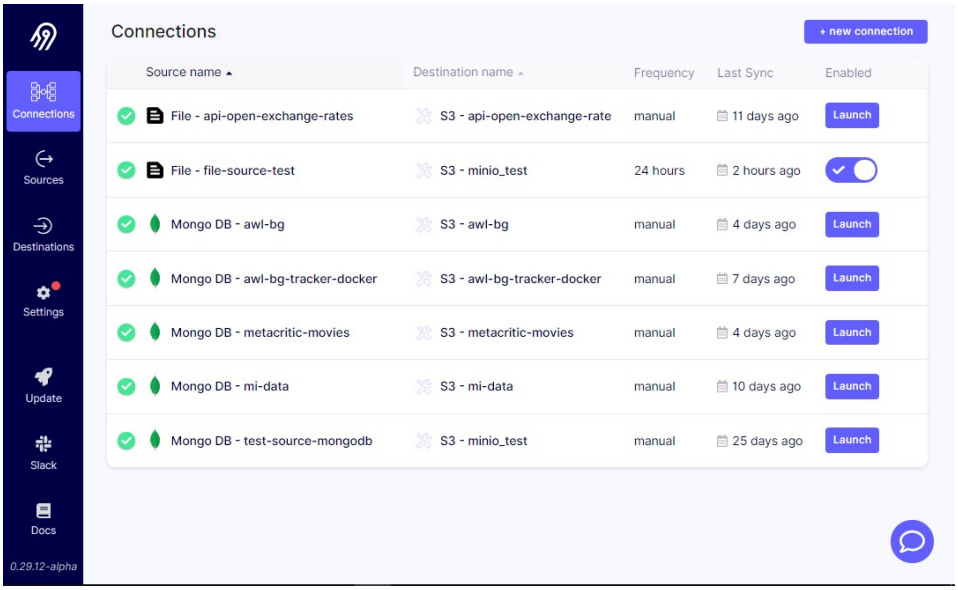
Airbyte é um projeto de código aberto para ingestão e integração de dados de diversas origens e destinos (AIRBYTE, 2021). O diferencial do projeto é de se posicionar como uma unidade centralizadora dos dados de entrada, garantindo maior controle e segurança sobre os processos de ingestão.

Para tal, conta com uma diversa gama de conectores de origem e destino desenvolvidos, tanto oficiais do projeto como desenvolvidos pela comunidade, su-

portando também conectores personalizados desenvolvidos como imagens Docker, no qual qualquer linguagem moderna pode ser utilizada.

O Airbyte conta com uma interface Web onde é possível visualizar, criar, atualizar e deletar seus componentes, assim como monitorar, agendar ou disparar os processos de integração. Existe ainda uma API, que suporta todas as operações que podem ser realizadas via interface Web. A Figura 2.8 ilustra uma tela da aplicação Web que exibe a listagem das conexões da dados existentes.

Figura 2.8 – Captura de tela da interface Web do Airbyte que exibe a listagem das conexões existentes



Fonte: Imagem do Autor

Os componentes do Airbyte envolvidos em uma ingestão de dados são: conectores-fonte, conectores-destino e conexões de dados.

2.6.1 Conectores-fonte

Os principais conectores-fonte existentes contemplam bancos de dados relacionais (e.g., Postgres e MSSQL) e não relacionais (e.g., MongoDB), arquivos (via HTTPS, SFTP, SCP, arquivos locais, etc.), serviços Web (e.g., Google Sheets

e GitHub) e os principais armazenamentos de objetos em nuvem (e.g., AWS S3 e Azure Blob Storage).

Ao consumir os dados de uma fonte, o Airbyte identifica seu esquema de dados e o armazena para ser replicado no destino.

2.6.2 Conectores-destino

Os conectores-destino são reduzidos em quantidade, quando comparados aos conectores-fonte, porém incluem as principais tecnologias que constituem os *data lakes* ou *data warehouses* atuais, como bancos relacionais, *data warehouses* na nuvem (e.g., Redshift e Snowflake), plataformas de *streaming* (e.g., Apache Kafka e Google Pubsub), armazenamentos de objetos em nuvem ou arquivos locais.

Para a escrita de arquivos, é possível configurar a extensão, bem como algoritmos de compressão utilizados.

2.6.3 Conexões de dados

As conexões de dados são as unidades que associam conectores-fonte com conectores-destino, definem o esquema e a forma de atualização dos dados e agenda suas execuções.

O esquema de dados identificado no conector-fonte é aplicado como padrão para o conector-destino. Contudo, é possível editá-lo para adequar nomenclatura ou estrutura.

Dependendo do conector-destino utilizado, os dados podem ser atualizados com as seguintes estratégias: *full refresh overwrite*, onde todos os dados de origem são processados e sobrescrevem o dado de destino; *full refresh append*, onde todos os dados de origem são processados, mas não sobrescrevem o dado de destino; e *incremental append*, onde, baseado em uma coluna chave definida

no esquema, apenas os dados novos (i.e., chaves existem na origem mais não no destino) são carregados.

2.7 MinIO

O MinIO é uma plataforma de armazenamento de objetos de alto desempenho, de código aberto (MINIO, 2021). Sua interface de acesso é compatível com a do serviço de armazenamento de objetos da AWS (Amazon Web Services), o S3, tornando-o acessível por uma vasta gama de serviços locais e de nuvem. É altamente escalável por ter suporte nativo à execução distribuída em contêineres Docker e Kubernetes.

Possui uma interface Web com recursos de controle de acesso, replicação, pré-visualização de dados e auditoria.

2.8 Apache Spark

Apache Spark é um *framework* de código aberto para processamento de grandes massas de dados em lotes ou em fluxo em alta performance, aplicações de aprendizado de máquina e processamento de grafos (SPARK, 2021).

Apesar de sua arquitetura complexa, o Spark possui APIs de alto nível de abstração em Java, Scala, Python, R e SQL, tornando o desenvolvimento facilitado e flexível.

Spark pode ser executado localmente, em um *cluster standalone*, para fins de desenvolvimento ou soluções de escopo reduzido, mas também é altamente escalável, tendo serviços que o suportam nas principais plataformas de nuvem. Além disso, é a ferramenta base para muitas tecnologias de processamento de dados, sendo a principal delas o Databricks (DATABRICKS, 2021).

2.9 Delta Lake

O Delta Lake é um *framework* que implementa uma camada de controle adicional aos serviços de armazenamentos de objetos, portando aspectos presentes em *data warehouses* para os *data lakes*, tais como transações ACID e verificação de esquemas de dados, os transformando em *data lakehouses* (ARMBRUST et al., 2020).

O *framework* propõe um pseudo formato de arquivos denominado *delta table* que consiste em um conjunto de arquivos, no formato *Parquet*, contendo os dados versionados em adição com arquivos JSON, que contém os *logs* das transações aplicadas sobre os dados . As *delta tables* são completamente compatíveis com a API do Spark, portanto é possível aplicar quaisquer transações suportadas, através de um *job* Spark, seja ele para processamento em lotes ou em fluxo.

As transações ACID são características conhecidas dos principais bancos de dados relacionais e trazem a consistência, integridade e segurança para transações concorrentes. Aplicadas ao Delta Lake, essas transações permitem que transações advindas de processamentos em lote ou em fluxo atuem sobre os mesmos dados, na mesma estrutura unificada e de forma paralela.

Outro aspecto inspirado nos sistemas de dados transacionais é a verificação de esquemas de dados. No Delta Lake, as *delta tables* impedem a escrita de dados de esquemas diferentes dos existentes, a não ser que seja explicitamente definido.

Uma vez que os dados são armazenados versionados a cada transação, é possível retomar a um determinado estado com intuito de correção ou para reproduzir um experimento específico. É ainda possível realizar auditorias sobre cada modificação realizada sobre os dados ou esquemas.

2.10 Dremio

O Dremio (DREMIO, 2021) é uma plataforma de código aberto que inclui uma camada intermediária de virtualização de dados entre as fontes de dados e as ferramentas de inteligência de negócios. Esta camada tem como objetivo unificar o acesso aos dados de diversas fontes em uma única unidade sem que haja a necessidade de se conhecer a estrutura em que o dado é fisicamente armazenado (LANS, 2012).

O Dremio é desenvolvido baseado no Apache Arrow, que é uma ferramenta também de código aberto para manipulação de dados em memória com alto desempenho através da linguagem SQL. Possui suporte para uma vasta gama de fontes de dados, incluindo bancos de dados relacionais (e.g., PostgreSQL), bancos de dados não-relacionais (e.g., MongoDB) e sistemas de arquivos distribuídos (e.g., MinIO) com suporte a diversas extensões, como Delta Lake.

Resultados de consultas executadas pelo Dremio podem ser persistidos em uma camada de cache com intuito de melhoria de tempo de resposta.

Além de uma interface Web para configuração das fontes de dados, consultas, persistências e definição de governança de acessos, o Dremio oferece uma API REST que contempla boa parte dessas operações. Além disso, existe um *plugin* que fornece conectividade através do protocolo ODBC, tornando o Dremio acessível por mais plataformas.

2.11 Apache Superset

O Apache Superset (SUPERSET, 2021) é uma plataforma de código aberto para exploração e visualização de dados através de gráficos e relatórios customizáveis.

A plataforma conta com uma interface Web onde é possível explorar os dados, criar relatórios e gráficos, exportar visões de dados específicas, conectar a

novas fontes de dados, controlar acessos e parâmetros de segurança, entre outras ações. Além disso, o Superset conta com uma API REST que expõe as principais funcionalidades da interface Web.

Por ser desenvolvido majoritariamente em Python, o Superset torna, por exemplo, o desenvolvimento de *plugins* de visualização personalizados mais fácil e acessível.

Em relação às fontes de dados, existem diversos conectores para uma considerável gama de bases de dados, incluindo Dremio, PostgreSQL e Apache Spark SQL.

2.12 Gatling

Gatling (GATLING, 2021) é uma ferramenta de código aberto para testes de cargas contínuos com integração a ferramentas de *pipeline* de DevOps. Os testes são definidos via código, na linguagem Scala. É possível realizar testes de desempenho e estresse em sistemas de software através de chamadas a seus recursos (e.g., por uma API REST) com objetivo de entender o alcance do sistema testado e identificar *endpoints* de gargalo.

Gatling tem uma interface Web para análise dos resultados obtidos através de relatórios gerados.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta análises de casos de implementações de armazenamento e gerenciamento de dados em um contexto de *Big Data*.

3.1 *International network performance and security testing based on distributed abyss storage cluster and draft of data lake framework*

O trabalho (CHA et al., 2018) propõe um *framework* para *Data Lake* baseado no projeto, de código aberto, Ceph, com alta escalabilidade horizontal e com mitigações para o problema conhecido como "*garbage dump*", i.e., quando os dados são levados para o *Data Lake*, porém nunca são analisados ou consumidos. Para contornar esse problema, os autores empregam análise topológica de dados (*Topological Data Analysis*, ou TDA, em inglês) e aprendizado de máquina.

Através de aplicação de TDA, que é um ramo da topologia matemática, tem-se um "formato" do conjunto de dados de entrada (i.e. *Data Lake*), propiciando o estudo de características como conectividade e compacticidade destes dados.

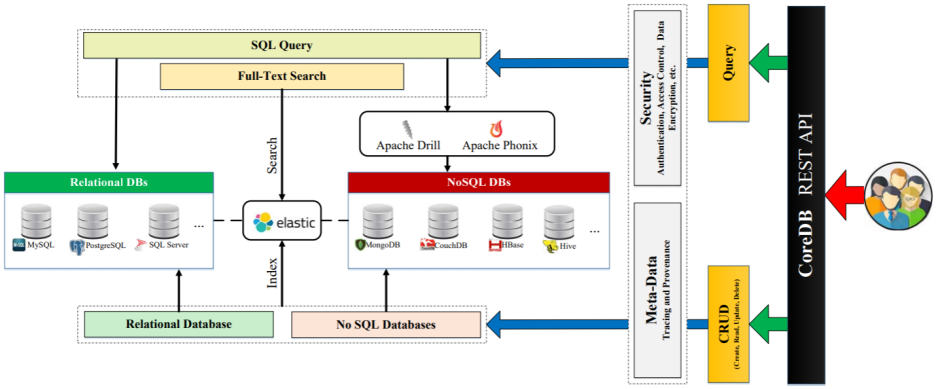
Algoritmos de aprendizado de máquina são aplicados para descoberta de padrões nos dados do conjunto .

O trabalho traz uma abordagem robusta, com viés performático, principalmente nas operações de leitura e escrita, porém com arquitetura menos preocupada com a versatilidade de tipos de dados de entrada ou com versionamento, catalogação e evolução dos dados. Além disso, apesar de ser projetado para ser configurado em equipamentos de *hardware* comuns, a configuração do ambiente com Ceph (CEPH, 2021) requer máquinas dedicadas para, pelo menos, os nós de armazenamento (OSDs). Isso pode ser um dificultador para contextos de pequenos projetos com limitações de gastos e difere do objetivo deste presente documento, o qual é proporcionar uma opção de processamento de dados consistente que utilize equipamentos de *hardware* pessoais.

3.2 CoreDB: *A Data Lake Service*

O trabalho (BEHESHTI et al., 2017), apresenta o CoreDB: um *framework*, de código aberto, para construção de *data lakes* que oferece aos desenvolvedores/-pesquisadores uma REST API segura para organização, indexação e consulta de dados e metadados advindos de múltiplas fontes de dados (de bancos de dados relacionais e não-relacionais). A arquitetura do *framework* de dados é ilustrada na Figura 3.1.

Figura 3.1 – Arquitetura CoreDB



Fonte: (BEHESHTI et al., 2017)

A motivação do projeto foi disponibilizar um banco de dados como um serviço (em inglês, *database-as-a-service*) para o desenvolvimento de aplicações Web orientadas a dados, fornecendo uma interface de dados no formato de arquivo JSON, que é formato disseminado em aplicações desta natureza.

O conceito de mais alto nível do CoreDB é o *Data Lake*: uma coleção de *Datasets* que, por sua vez, são conjuntos de Entidades (i.e. dados provenientes de bancos de dados relacionais e/ou não-relacionais). Cada *Dataset* representa a conexão do CoreDB com um segmento (e.g., uma tabela em um banco relacional) de um banco de dados físico. As entidades, por fim, representam os dados em si e são persistidas em formato JSON dentro do CoreDB.

Através da REST API, é possível aplicar operações de criação, leitura, atualização e exclusão de *Data Lakes*, *Datasets* e Entidades. O CoreDB possibilita pesquisa de texto completo (em inglês, *Full-text search*), com indexação automática no Elasticsearch, ao criar um novo dataset. As transações suportam os princípios ACID e são garantidas, inclusive para os bancos de dados não relacionais. Além disso, são suportados *joins* entre dados de fontes e naturezas diferentes.

Em termos de segurança, o CoreDB possui autenticação com diferentes níveis de acesso ao sistema e aos objetos. Ademais, provê um grafo de proveniência que armazena e associa usuários, ações realizadas (e.g., criação e leitura) e entidades para futuras auditorias.

Em comparação com o projeto deste presente documento, o CoreDB possui vantagens relevantes, como a aplicação de segurança em um nível mais granular (i.e., segurança por objeto) e uma REST API unificada. Contudo, apresenta algumas limitações consideráveis como o uso exclusivo do formato JSON, que possui baixo desempenho de leitura e escrita quando comparado com, respectivamente, Apache Parquet (MORO, 2020) e Apache Avro (MAEDA, 2012); a impossibilidade de ingestão de dados não estruturados (e.g., imagens e videos); e a inexistência de um mecanismo de versionamento e recuperação de dados (i.e., não é possível, a nível de aplicação, recuperar dados excluídos ou sobrescritos).

3.3 *Hardware and software data processing system for research and scientific purposes based on Raspberry Pi 3 microcomputer*

O trabalho (PANKOV; NIKIFOROV; DROBINTSEV, 2020) propõe uma solução de baixo custo, com a utilização de computadores de placa única (em inglês, *single-board-computer* ou SBC), para processamento de grande massas de dados para contextos científicos e de pesquisa. Outra importante contribuição do estudo é o levantamento comparativo de custos e efetividade de implementações de *clusters* baseados em computadores Raspberry Pi com máquinas *desktop* pessoais.

A solução proposta consiste em um *cluster* Hadoop heterogêneo implementado utilizando 4 computadores Raspberry Pi de modelos distintos. Os autores executam testes básicos de desempenho, através de operações de MapReduce, considerando diferentes volumes de dados, parâmetros de configuração, quantidade de nós (i.e., quantidade de computadores ativos) e atribuições de cada nó (i.e., nó *master*, que distribui tarefas e monitora outros nós, ou nó *worker*, que processa os dados).

As conclusões obtidas são: o *cluster* apresenta, como esperado, efetivo desempenho em tarefas que podem ser fragmentadas em um grande número de pequenas cargas, em detrimento a grandes tarefas não paralelizáveis; ao elevar o número de nós, nota-se também a elevação da ocorrência de erros de alocação de memória nos nós *workers*, sendo necessário ajuste das configurações do Hadoop para uso mais adequado dos recursos dos SBCs, de acordo com cada topologia aplicada ao *cluster*.

Os estudos levantados evidenciam que *clusters* formados por computadores Raspberry Pi oferecem considerável redução de custos, baixo consumo energético e ao mesmo tempo recursos semelhantes a máquinas pessoais, podendo inclusive superá-las em contextos de processamento de grandes montantes de dados (SRINIVASAN et al., 2018).

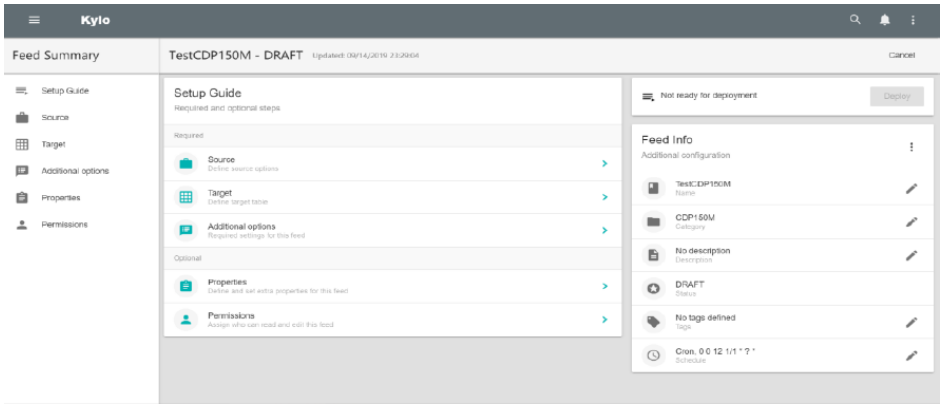
Os autores trazem uma solução limitada no sentido de recursos de dados, não tratando especificamente da organização, processamento, qualidade ou governança destes, porém abordando uma possibilidade interessante de infraestrutura com escalabilidade horizontal, baseada em projetos de software e hardware abertos e de baixo custo de aquisição e consumo energético. O trabalho deste presente documento busca atrelar o custo acessível e a flexibilidade desse projeto com práticas e ferramentas de código aberto disseminadas no mercado para fornecer uma interface de dados confiável e robusta para pesquisadores, cientistas e engenheiros.

3.4 *Kylo Data Lakes Configuration deployed in Public Cloud environments in Single Node Mode*

Apesar de *Data Lakes* serem atualmente amplamente implementados em ambientes empresariais, muitos ainda os desconhecem ou não conseguem os descrever corretamente. O trabalho de Peng (2019) faz, inicialmente, uma revisão sobre o conhecimento já publicado nessa área e, posteriormente, foca em descrever a arquitetura e configuração de um *framework* para *Data Lakes* denominado Kylo. Com o objetivo de avaliar o desempenho desse projeto, é conduzido um experimento focado na ingestão de dados sob diferentes cargas e formatos de dados.

Kylo é uma plataforma de código aberto para *Data Lakes*, desenvolvida pela Teradata Company, adequada para estudos acadêmicos e que baseia-se em uma gama de projetos abertos para prover recursos de ingestão, preparação, monitoramento, descoberta e segurança de dados. O projeto conta com uma camada de aplicação Web, ilustrada na Figura 3.2, que oferece recursos focados nos usuários de negócio, incluindo analistas, cientistas e administradores de dados. A plataforma pode ser configurada em ambientes locais ou ambientes de nuvem pública ou privada, sendo compatível com Cloudera, HortonWorks, Amazon EMR, entre outros serviços.

Figura 3.2 – Interface da aplicação Web do Kylo



Fonte: (PENG, 2019)

Peng conduz o experimento em três etapas. A primeira consiste na realização da carga de arquivos, um a um, de formato CSV (do inglês, *comma separated value*) e com diferentes tamanhos - 1 KB, 10 MB, 50MB e 100 MB. Na etapa posterior, são realizadas tentativas de cargas de múltiplos arquivos de dados de uma vez (i.e., em lotes). Por fim, são carregados arquivos compactados - formato ZIP - no *Data Lake*.

Pelos resultados do experimento, é concluído que apesar do projeto ser, pelas palavras do autor, "poderoso e criativo" e de pequenas cargas de arquivos menores funcionarem corretamente, existem consideráveis limitações para alguns cenários encontrados. Há uma limitação de 58 MB por arquivo carregado, não sendo possível, portanto, ingerir arquivos únicos maiores que esse valor. A plataforma não consegue gerenciar a carga de múltiplos arquivos em lotes, causando erros e resultados inesperados. Ademais, segundo Peng, a configuração do ambiente é surpreendentemente complexa e custosa.

O projeto Kylo tem muitos aspectos semelhantes ao projeto do atual documento, compartilhando, inclusive, algumas mesmas soluções, como o Apache Spark e PostgreSQL em suas arquiteturas.

Os diferenciais positivos da plataforma da Teradata são: a interface de usuário amigável, trazendo facilidade na utilização dos módulos e recursos por diferentes perfis de usuários; a compatibilidade já existente para implantação em ambientes de nuvem; e a segurança mais granular - a nível de objeto - dos dados. Por outro lado, além das limitações já elucidadas pelo autor, o Kylo possui restrições de *hardware* - mínimo de 16 GB RAM - e *software* - apenas as distribuições Linux SUSE, Ubuntu e CentOS são suportadas.

3.5 *Data Lake Lambda Architecture for Smart Grids Big Data Analytics*

O trabalho (MUNSHI; MOHAMED, 2018) apresenta e implementa um ecossistema de *Big Data* em nuvem como uma alternativa para o armazenamento,

processamento e visualização da grande massa de dados provenientes do setor da energia, que possa ser utilizado por diversos grupos de usuários, incluindo pessoas não técnicas. O ecossistema se baseia no conceito de arquitetura *Lambda* para gerenciar dados de produção energética em lotes e em tempo real, advindos dos chamados *Smart Grids*.

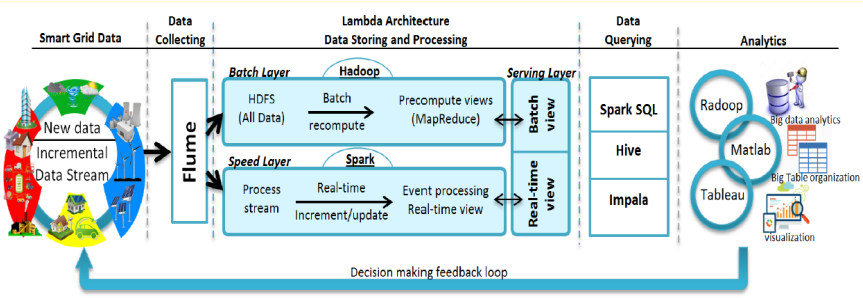
O modelo tradicional centralizado de geração e coleta dos dados energéticos vem sendo substituído por uma nova abordagem denominada *Smart Grids*, que é uma alternativa descentralizada, de maior cobertura e com a geração de dados de maior volume, velocidade (i.e., dados que eram gerados mensalmente, passam a ser gerados no intervalo de poucos minutos) e variedade (diferentes tipos de dados são gerados de várias fontes).

O conceito de arquitetura *Lambda* (WARREN; MARZ, 2015) tem como objetivo a construção de uma arquitetura de dados que seja capaz de computar funções arbitrárias em conjuntos de dados distribuídos em tempo real, mas também de combinar capacidades de processamento em lotes e em tempo real para equilibrar a latência de dados e a tolerância a erros.

Tal arquitetura, com implementação ilustrada na Figura 3.3, consiste em 3 camadas principais: a *Batch layer*, com as atribuições de armazenar a crescente massa de dados distribuídos e pré-computar *views* em lotes desses dados; a *Speed layer* que usa uma abordagem incremental para a pré-computação de *views*, considerando apenas os dados mais recentes; e a *Serving Layer* que consiste em uma base de dados distribuída que indexa as *views* em lotes para serem consumidas com baixa latência e mescla os resultados das computações das camadas *Batch* e *Speed*.

São realizados experimentos para avaliação de desempenho e capacidade de armazenamento, visualização e aplicação de algoritmos de clusterização sob a arquitetura implementada. Por meio da análise dos resultados, o autor constata a preocupação de que podem ocorrer sobrecargas anormais em alguns nós,

Figura 3.3 – Arquitetura do ecossistema Smart grid Big data



Fonte: (MUNSHI; MOHAMED, 2018)

indicando que seria interessante considerar a aplicação de alguma estratégia para balanceamento de cargas entre os nós. Contudo, é observada robustez nos resultados obtidos, inclusive sob situações simuladas de falhas nos nós e desconexões de rede.

Existem consideráveis semelhanças entre o projeto de Munshi e o trabalho proposto por este atual documento. A principal delas é o uso de mecanismos de consulta de dados, via SQL, ao *Data Lake* para geração de visualizações de dados e a expansibilidade para a execução de aplicações de *Machine Learning* no topo das arquiteturas apresentadas.

Entretanto, existem aspectos que favorecem o trabalho de Munshi, como a escalabilidade horizontal facilitada pelo uso nativo do ecossistema Hadoop; a maior tolerância a falhas de *hardware* e a acessibilidade dos endereços IP dos nós pela Internet, por serem instanciados em uma nuvem pública.

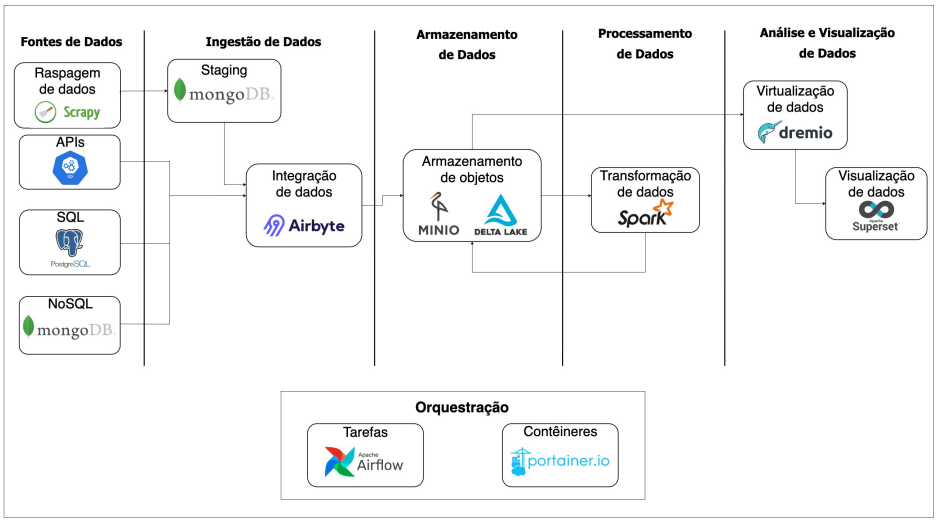
O trabalho deste atual documento também apresenta diferenciais, tais como a capacidade de versionamento de dados, flexibilidade para orquestração de rotinas customização para aquisição de dados (e.g., raspadores de dados), o uso do Apache Spark para processamento de dados em lotes, que pode reduzir de 10 a 100 vezes o tempo de execução, comparado ao MapReduce (GOPALANI; ARORA, 2015) e a segurança de se executar os processos localmente sem ter os dados trafegados pela Internet.

4 METODOLOGIA

4.1 Arquitetura proposta

A arquitetura projetada para atender os objetivos deste trabalho é apresentada na Figura 4.1. A arquitetura é segmentada por camadas, que são detalhadas nas subseções de mesmo nome a seguir.

Figura 4.1 – Arquitetura proposta



Fonte: Imagem do Autor

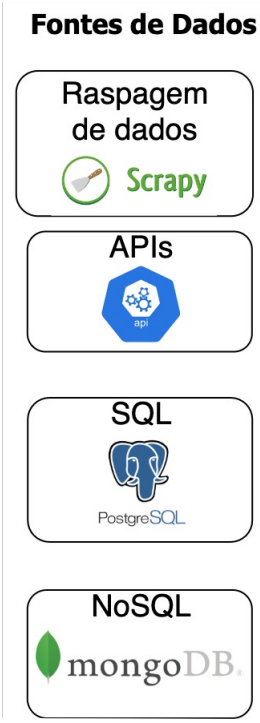
4.1.1 Fontes de dados

A plataforma contempla 4 (quatro) grupos de fontes de dados suportadas, como ilustrado na Figura 4.2: bancos de dados relacionais, com suporte a PostgreSQL, MySQL, SQL Server e Oracle DB; bancos de dados não relacionais, com suporte a Mongo DB; APIs (disponibilizadas via protocolo HTTP/HTTPS) e raspagens de dados customizadas.

Através dessas fontes, é possível consumir dados de diversas naturezas, sejam eles estruturados, semiestruturados ou não estruturados, como imagens. O

suporte fornecido para o desenvolvimento de raspagens de dados abrange e facilita a coleta de uma enorme gama de dados.

Figura 4.2 – Camada de fontes de dados



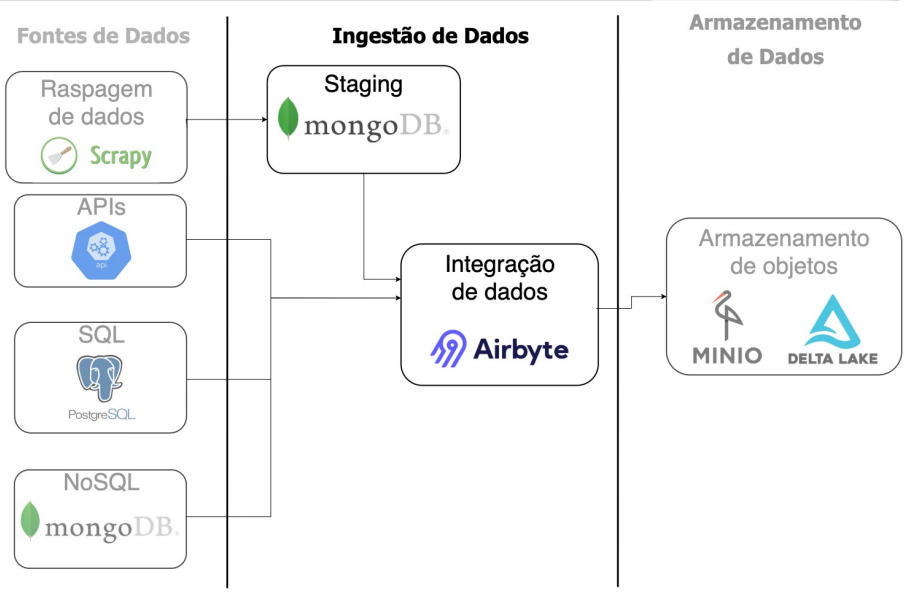
Fonte: Imagem do Autor

4.1.2 Ingestão de dados

A camada de ingestão de dados tem os componentes e estrutura ilustrados na Figura 4.3.

Existem dois fluxos para a ingestão de dados, dependendo de qual é a fonte de dados. Contudo, todos os dados ingeridos são armazenados, ao final do fluxo, no armazenamento de objetos (i.e., MinIO).

Figura 4.3 – Camada de ingestão de dados



Fonte: Imagem do Autor

4.1.2.1 Fluxo de ingestão por raspagem de dados

Para dados advindos de raspagem de dados o fluxo é definido pelos seguintes passos:

1. *Implementação do raspador de dados:* Definição de uma imagem Docker, baseada em quaisquer linguagens de programação suportadas e utilizando o modelo de Dockerfile do Apêndice C, que capture os dados de uma fonte de dados acessível e os escreva na instância do MongoDB em execução. Para aplicações desenvolvidas em Python, existe um modelo proposto, apresentado no Apêndice D.

2. *Arquivo de descrição de fonte de dados*: Definição de um arquivo JSON que descreva, segundo um modelo pré-definido no Apêndice E, a fonte de dados a ser utilizada e seus parâmetros. No caso da raspagem de dados, a fonte de dados sempre será a instância do MongoDB, logo é necessária a definição deste arquivo apenas antes da primeira execução.

3. *Arquivo de descrição de destino de dados*: Definição de um arquivo JSON que descreva, segundo um modelo pré-definido no Apêndice F, o destino de dados a ser utilizado e seus parâmetros. Como todos os dados ingeridos pela plataforma têm o mesmo destino (i.e., MinIO), esse arquivo é definido apenas um única vez.

4. *Arquivo de descrição de conector do Airbyte*: Definição de arquivo JSON para descrição dos parâmetros mandatórios de uma conexão do Airbyte (i.e. fonte, destino e modo de execução), bem como os opcionais, seguindo o modelo do Apêndice G.

5. *Implementação da DAG Airflow*: Escrita de um arquivo Python que descreva uma DAG que defina e organize as etapas supracitadas. O Apêndice H define um modelo para definições de dags de ingestão. O modelo se utiliza de uma biblioteca mínima desenvolvida para consumo da API do Airbyte, disponível no repositório do projeto. As tarefas mandatórias que uma DAG de ingestão deve contemplar, em ordem, são:

    (a) Criação dos objetos que definem as fontes, destinos e conexões de dados no Airbyte, caso não existam;

    (b) Execução e monitoramento da integração de dados no Airbyte;

    (c) Limpeza dos dados temporários da camada de *staging* na instância do MongoDB.

4.1.2.2 Fluxo de ingestão para as demais fontes de dados

Para os dados advindos de bancos de dados relacionais, não relacionais e APIs, são aplicadas as mesmas etapas 2, 3, 4 e 5 do fluxo de raspagem de dados. No entanto, para a etapa 5, o item *(c)* (i.e. limpeza da camada *staging*) não existe.

4.1.3 Armazenamento de Dados

Os componentes que formam a camada de armazenamento de dados estão ilustrados na Figura 4.4.

Todos dados da plataforma são centralizados na instância do MinIO, constituindo um *Data Lakehouse* através da aplicação da camada de abstração do Delta Lake sob os objetos armazenados, em uma determinada modelagem de dados.

4.1.3.1 Modelagem *Data Lakehouse*

A modelagem dos dados armazenados no MinIO é dividida em 3 (três) camadas:

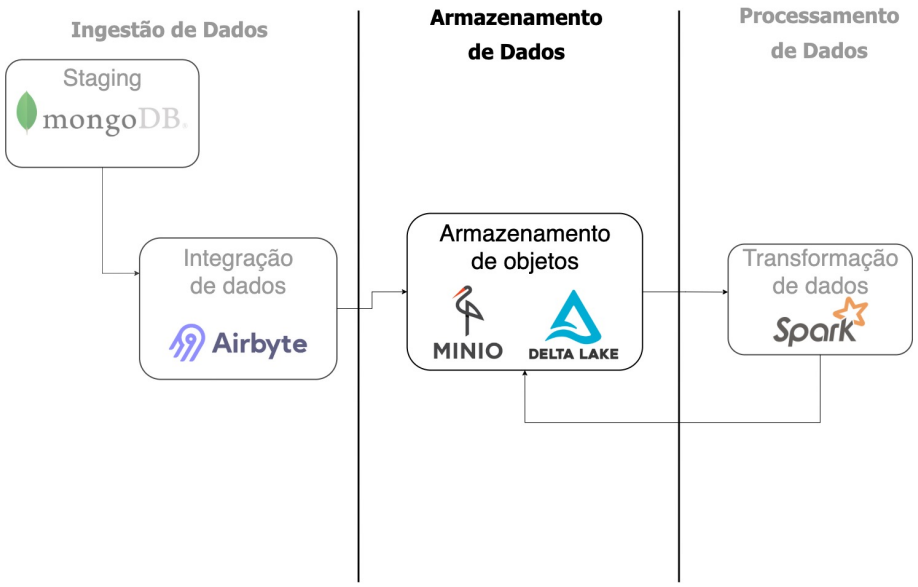
- *01_bronze*: Camada que contempla os dados da forma em que são ingeridos, no mesmo formato de arquivo e sem transformações de negócio, qualidade de dado ou tratamento de obscuridade de dados sensíveis. Nenhum dado é deletado ou alterado nessa camada;

- *02_silver*: Camada com dados ainda sem teor de negócio, porém com aplicação de filtros (e.g., remoção de duplicatas indesejadas), limpo (e.g. correção de tipagem e formato de dados) e enriquecido (e.g., junção de dados correlatos). É formada exclusivamente através do consumo dos dados da camada *01_bronze*;

- *03_gold*: Camada com aplicação de regras de negócio nos dados e, normalmente, com modelagem dimensional (i.e., com dimensões e fatos), em uma estrutura que prioriza o desempenho de leitura dos mesmos por ferramentas clientes (e.g., Dremio e Apache Superset). É formada exclusivamente através do consumo dos dados da camada *02_silver*.

Todos os dados das camadas *02_silver* e *03_gold* são escritos como *delta tables* e versionados a cada nova atualização.

Figura 4.4 – Camada de armazenamento de dados



Fonte: Imagem do Autor

4.1.4 Processamento de dados

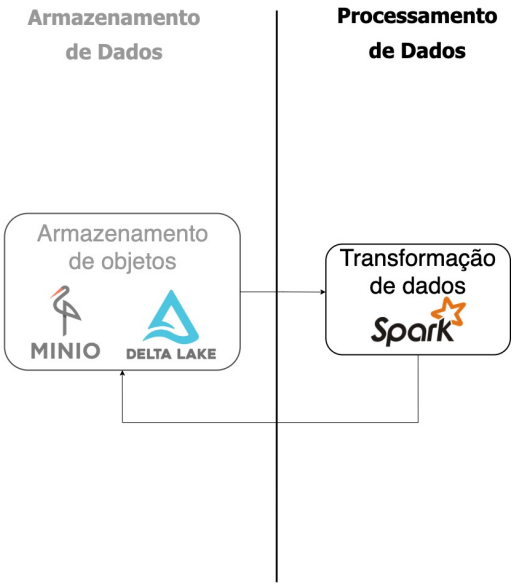
A camada de processamento de dados é composta por um *cluster* Apache Spark *standalone* e se posiciona na arquitetura como o mostrado na Figura 4.5, lendo dados do *Data Lakehouse* e os devolvendo transformados.

O fluxo para aplicação das transformações é dados pelas seguintes etapas:

1. *Script de transformação de dados*: Implementação de um arquivo Python que define um *job* a ser executado pelo *cluster standalone*, seguindo o modelo apresentado no Apêndice I.

2. *Implementação da DAG Airflow*: Escrita de um arquivo Python que descreva uma DAG que execute e agende as execuções dos *scripts* de transformação desenvolvidos. Cada *script* é submetido ao cluster Apache Spark com os parâmetros de execução e configuração, como o exemplificado no Apêndice J.

Figura 4.5 – Camada de processamento de dados



Fonte: Imagem do Autor

4.1.5 Análise e visualização de dados

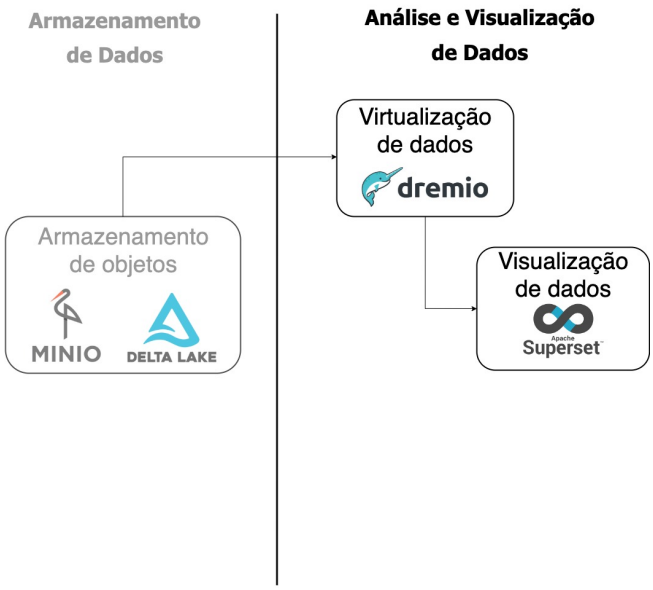
A camada que engloba Análise e visualização de dados é ilustrada pela Figura 4.6 e é composta por uma instância do Dremio em conjunto com uma instância do Apache Superset, responsáveis pela virtualização e visualização de dados, respectivamente.

4.1.5.1 Virtualização de dados

A virtualização de dados possibilita a centralização de acesso e consulta a fontes de dados físicos diversas. Dessa forma, o Dremio, que possui conectores em diversas soluções de visualização de dados, atua como um intermediário entre estas e as diversas fontes de dados.

Outra característica dessa arquitetura é a possibilidade de se experimentar novos dados (e.g., planilhas eletrônicas), adicionando-os no virtualizador e fazendo a junção com os dados advindos das camadas físicas.

Figura 4.6 – Camada de análise e visualização de dados



Fonte: Imagem do Autor

As virtualizações do projeto são modeladas em 3 (três) camadas hierárquicas:

- *Staging*: camada na qual são definidas as conexões com os dados físicos, sem quaisquer alterações.
- *Business*: através da leitura do consumo da camada *Staging* é gerada a camada *Business*, onde são unidos vários conjuntos de dados que dizem respeito a um projeto em específico.
- *Application*: camada que prepara os dados para consumo para uma visão em específico. É formada a partir da camada *Business*.

A persistência de dados como replicações, resultados de *jobs* e *upload* de conjuntos de dados de experimentação é feita de forma distribuída, conectado à instância do MinIO (i.e., na máquina mestre). Os metadados, como definição de

conexões, dados de usuários e segurança são persistidos localmente na máquina que executa a instância do Dremio (i.e., máquina trabalhadora).

4.1.5.2 Visualização de dados

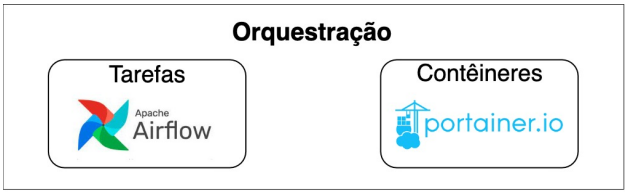
A visualização de dados da plataforma se baseia na criação de gráficos e relatórios no Apache Superset, a partir dos dados da camada de virtualização (i.e., Dremio). Cada visual é desenvolvido diretamente na aplicação Web, de forma individual, tendo suporte para configuração de comportamentos como periodicidade de atualização dos dados e exportação e envio agendado de relatórios, como figuras, via e-mail.

Foi implementado um *script* de inicialização para mapear a instância do Dremio como uma fonte de dados do Apache Superset, um modelo é mostrado no Apêndice K.

4.1.6 Orquestração

A camada de orquestração do projeto é aplicada a dois níveis de arquitetura, contêineres e tarefas, através de instâncias do Apache Airflow e Portainer, respectivamente.

Figura 4.7 – Camada de orquestração de tarefas e contêineres



Fonte: Imagem do Autor

4.1.6.1 Orquestração de contêineres

Cada aplicação pertencente a plataforma proposta, incluindo o próprio Portainer, possui uma imagem Docker definida. Analogamente, cada camada da

arquitetura possui uma *stack* Docker, definida por um arquivo *docker-compose.yml* que utiliza essas imagens, configurada no Portainer.

Visando otimizar o consumo de memória e processamento da máquina trabalhadora, é implementada uma biblioteca para consumo da API REST do Portainer, responsável por instanciar as *stacks*, sob demanda, bem como encerrá-las após o processamento de cada uma. A classe principal da biblioteca é ilustrada no Apêndice L.

Dessa forma, são implementadas DAGs no Apache Airflow para orquestrar os contêineres de acordo com cada tipo de execução a ocorrer. Por exemplo, antes de iniciar o processamento de uma ingestão, é instanciada a *stack* referente à camada de ingestão e, após a finalização do carregamento dos dados, a *stack* é finalizada.

4.1.6.2 Orquestração de tarefas

O Apache Airflow é instanciado de forma distribuída nas máquinas mestre e trabalhadora. Na primeira, são instanciados o agendador, o banco de metadados e a interface Web, enquanto na segunda, apenas o executor Celery, que é o componente de maior consumo de *hardware*.

Nessa arquitetura, novas máquinas trabalhadoras podem ser adicionadas, instanciando novos executores Celery e escalando horizontalmente a plataforma.

4.2 Infraestrutura do projeto

A plataforma é dimensionada para projetos de baixo custo e com volume de carga reduzido. Dessa forma, a infraestrutura do projeto é definida por 2 (duas) máquinas: *máquina mestre* e *máquina trabalhadora*.

4.2.1 Máquina mestre

A máquina mestre é a responsável por duas principais tarefas: servir o orquestrador da plataforma (i.e., Apache Airflow) e fornecer uma solução de armazenamento centralizada. É implementada por um computador de baixo custo, podendo ser utilizado um microcomputador popular, como um Raspberry Pi 3 Model B[1]. O sistema operacional utilizado é o Raspberry Pi OS Lite 5.10. As aplicações provisionadas na máquina mestre estão listadas na Tabela 4.1.

Por centralizar o armazenamento, a máquina mestre deve ser alcançável pela máquina trabalhadora, de forma que seja possível ler e escrever dados (i.e., MinIO e PostgreSQL) e mensagens (i.e., Redis) a partir dos processos dessa última.

Tabela 4.1 – Aplicações provisionadas na máquina mestre

Aplicação	**Porta de serviço**
Redis	6379
MinIO (UI)	9000
MinIO (API)	9001
PostgreSQL	5432
Apache Airflow (scheduler)	Aleatória
Apache Airflow (UI)	8000

4.2.2 Máquina trabalhadora

A máquina trabalhadora é a responsável por executar as operações com dados na plataforma, sejam elas ingestão, transformação, análise ou visualização. É implementada em uma máquina de maior poder de processamento que a mestre, mas que pode ser utilizada uma máquina pessoal com configurações médias, como o disposto na Tabela 4.2, baseada nos requisitos mínimos do Apache Spark que é a aplicação mais onerosa da plataforma. O sistema operacional utilizado é

[1] https://www.raspberrypi.com/products/raspberry-pi-3-model-b/

o Windows 10 Enterprise 19043.1288 devido a disponibilidade da máquina utilizada. Contudo, a plataforma é compatível com outros sistemas operacionais (e.g., distribuições Linux) que tenham suporte ao Docker. As aplicações provisionadas na máquina trabalhadora estão listadas na Tabela 4.3.

Tabela 4.2 – Requisitos mínimos recomendados para máquina trabalhadora

Recurso	**Especificação mínima**
CPU (Cores)	8 cores
Memória RAM	8 GB
Rede	10 Gigabit

Tabela 4.3 – Aplicações provisionadas na máquina trabalhadora

Aplicação	**Porta de serviço**
Airbyte (servidor backend)	8001
Airbyte (servidor frontend)	8000
MongoDB	27017
Mongo Express	8017
Apache Spark (master)	7077
Apache Spark (UI)	8080
Apache Spark (worker #1)	8081
Apache Spark (worker #2)	8082
Dremio (UI)	9047
Dremio (ODBC)	31010
Superset	8088
Airflow (Celery worker)	8793
Airflow (servidor Flower)	5555

Apesar do dimensionamento proposto para duas máquinas, é possível escalar horizontalmente adicionando novas máquinas trabalhadoras.

4.3 Fora do escopo da solução

Estão fora do escopo da solução proposta por este trabalho:

- Utilização do recurso de armazenamento de objetos distribuídos do MinIO;

- Mapeamento automático de novos objetos do *Data Lakehouse* no Dremio utilizando a API REST do Dremio;
- Mecanismos de aprimoramento de criptografia de comunicação, tais como SSL (*secure socket layer*) aliado a um protocolo de autenticação de rede mais robusto, como o Kerberos (MIT, 2021);
- Mecanismos de unificação de autenticação sobre as aplicações envolvidas na plataforma, como o Apache Ranger (APACHE, 2021);
- Suporte para processamento de dados em tempo real ou *streaming* de dados, através de ferramentas como o Apache Kafka (KAFKA, 2021);
- Catalogação de dados para descoberta de dados e pesquisa de metadados, através de ferramentas como Amundsen (AMUNDSEN, 2021);
- Mecanismos de teste de dados, como o Great Expectations (EXPECTATIONS, 2021) ou Deequ (DEEQU, 2021);
- Inclusão de suporte a *pipelines* de aprendizado de máquina na plataforma, como o Kedro (KEDRO, 2021);
- Uso de Kubernetes (KUBERNETES, 2021) ao invés das imagens Docker para maior escalabilidade e elasticidade.

4.4 Conjuntos de dados utilizados

A fim de testar o funcionamento e a conexão entre os componentes da plataforma, são utilizados conjuntos de dados de diferentes origens e estruturas. Cada conjunto utilizado se enquadra em pelo menos um dos requisitos abaixo:

- Proveniente do raspador de dados da plataforma;
- Proveniente de uma API disponibilizada via HTTP/HTTPS;

- Proveniente de uma base de dados relacional;
- Proveniente de uma base de dados não relacional.

4.5 Código-fonte

Todo o código-fonte da solução está disponibilizado em 3 repositórios remotos: osdearc[2], osdearc-rpi[3] e osdearc-dags[4].

4.5.1 Repositório *osdearc*

Repositório remoto que contém todos os arquivos de configuração do que é denominado neste documento como "máquina trabalhadora". Inclui artefatos como *.Dockerfile*, *docker-compose.yml* e arquivos de configuração específica para cada aplicação que executará.

4.5.2 Repositório *osdearc-rpi*

Repositório remoto que contém todos os arquivos de configuração do que é denominado neste documento como "máquina mestre". Inclui artefatos como *.Dockerfile*, *docker-compose.yml* e arquivos de configuração específica para cada aplicação que executará.

4.5.3 Repositório *osdearc-dags*

Repositório remoto que contém todos os códigos-fonte que definem as DAGs que serão executadas na plataforma. Inclui arquivos Python para definição das DAGs, bem como as bibliotecas Python customizadas que sejam desenvolvidas para atender a DAGs.

[2] https://github.com/GabrielMMelo/osdearc/
[3] https://github.com/GabrielMMelo/osdearc-rpi/
[4] https://github.com/GabrielMMelo/osdearc-dags/

4.6 Instanciação da solução

Os repositórios *osdearc* e *osdearc-rpi* contam com um arquivo denominado *README.md* que contém um descritivo do projeto, bem como as instruções para instalação e instanciação da solução.

5 RESULTADOS

5.1 Testes de carga

Foram realizados testes de carga utilizando a ferramenta de código aberto Gatling. Para execução dos testes, foram configuradas chamadas a API do Apache Airflow no *endpoint* referente a DAGs criadas para ingestões de 5 (cinco) conjuntos de dados fictícios, criados utilizando a biblioteca Python *Faker*, com diferentes tamanhos: 1 megabyte, 10 megabytes, 100 megabytes, 1 gigabyte e 2 gigabytes. Para cada conjunto, foram realizadas 5 execuções e considerado o tempo médio de execução. A Tabela 5.1 mostra os resultados obtidos.

Tabela 5.1 – Resultados obtidos no teste de carga

Conjunto de dados	**Percentual de sucesso (%)**	**Tempo médio de execução (s)**
1 MB	100	3.1
10 MB	100	7.1
100 MB	100	40.7
1 GB	75	320.2
2 GB	0	-

Nenhuma das execuções do conjunto de dados de 2 GB teve êxito devido a estouros de memória na máquina mestre, decorrentes da limitação física de memória (i.e., 1 *gigabyte*) e da configuração de memória *swap* (i.e., 256 *megabytes*) da máquina mestre.

5.2 Custos

Os custos do projeto foram segmentados pelos custos de aquisição das máquinas utilizadas. Não foram considerados os gastos referentes ao consumo energético dos componentes, uma vez que o objetivo é a utilização de máquinas não dedicadas exclusivamente aos objetivos deste projeto.

Os valores dos componentes foram orçados fazendo a média dos menores preços encontrados em 3 grandes lojas digitais[1] de atuação no Brasil, na data de 06 de novembro de 2021.

5.2.1 Máquina mestre

A máquina mestre foi implementada utilizando um microcomputador com as especificações técnicas listadas na Tabela 5.2.

Tabela 5.2 – Especificações da máquina mestre

Componente	Valor médio (R$)
Raspberry Pi 3 model B	470,96
Micro SD Kingston classe 10 (32 GB)	52,86
Carregador micro USB (2A)	36,75

5.2.2 Máquina trabalhadora

A máquina trabalhadora foi implementada utilizando um computador pessoal com as especificações técnicas listadas na Tabela 5.3.

Tabela 5.3 – Especificações da máquina trabalhadora

Componente	Valor médio (R$)
Intel(R) Core(TM) i5-10400F	1.513,57
SSD XPG S41 TUF, 256GB, M.2	414,30
Memória Crucial Ballistix 16GB DDR4 2666 Mhz	623,41
Fonte Redragon Gc-ps002 600w	485,63

5.3 Avaliação qualitativa

De uma perspectiva didática, a plataforma cobre diversos estágios de *pipelines* de dados, fornecendo a possibilidade de experimentação de novas confi-

[1] *https://www.amazon.com.br*, *https://www.mercadolivre.com.br/* e *https://www.americanas.com.br/*

gurações, escalabilidade horizontal e aprendizado de ferramentas de amplo uso de mercado.

Apesar de não fomentar o conhecimento referente a serviços específicos de provedores de computação nuvem, enriquece o conhecimento de menor nível de abstração de máquina, com preocupações de alocação de memória, processos e rede.

6 CONCLUSÃO

Analisando os resultados obtidos nos testes de carga, é possível concluir que a plataforma possui limitações notáveis sobre o volume máximo de carga de dados (i.e., limite empírico de 1 GB por carga), mas que se adéqua às expectativas traçadas para projetos de escopos reduzidos.

Acerca do levantamento dos custos da plataforma, conclui-se que o investimento médio atual para as máquinas mestre e trabalhadora são de, respectivamente, R$560,57 e R$3.036,91. O valor total de R$3.597,48, que desconsidera os custos de manutenção e consumo energético, representa uma alternativa para instituições de ensino e estudantes pois a infraestrutura adquirida não é destinada a usos exclusivos ao projeto, isto é, podem ser usadas como máquinas pessoais para estudo, trabalho e lazer, desde que respeitados os requisitos mínimos necessários para a execução da plataforma.

De um ponto de vista da aplicação didática do projeto, a plataforma apresenta relevante cobertura de diversos estágios de *pipelines* de dados, fornecendo a possibilidades de experimentações de configurações, escalabilidade horizontal e de introdução à ferramentas de dados de amplo uso de mercado. Por ser um projeto com enfoque no aprendizado e em estágio inicial, sua complexidade é menor quando comparada com projetos de mercado, onde existem reais preocupações com segurança, disponibilidade, escalabilidade vertical, por exemplo. Outro aspecto favorecedor do potencial didático, é a modularização da arquitetura do projeto, na qual é possível estudar e implementar estágios do pipeline de dados de forma isolada.

Além dos itens levantados na Seção 4.3, o estudo de alternativas de hardware de melhor eficiência e do consumo energético das máquinas envolvidas são considerados como trabalhos futuros.

Durante o desenvolvimento e estudo envolvido neste trabalho de conclusão de curso, foram revisitados conceitos e problemas estudados, principalmente,

nas disciplinas de *Sistemas Distribuídos*, *Sistemas Gerenciadores de Bancos de Dados*, *Sistemas operacionais*, *Sistemas embarcados* e *Engenharia de Software*. O aprendizado adquirido neste percurso trouxe maior proximidade técnica e de negócio às frentes requisitadas no mercado, como engenharia de dados e ciência de dados. Ao curso de Ciência da Computação da Universidade Federal de Lavras, sugiro o fomento por iniciativas de ensino, pesquisa e extensão que alcancem o grande campo da engenharia de dados, promovendo formação aos egressos em uma posição que é um dos motores das ações de transformações digitais atuais.

REFERÊNCIAS

AIRBYTE. **Airbyte**. 2021. Disponível em: <https://airbyte.io/>.

AIRFLOW. 2021. Disponível em: <https://airflow.apache.org/>.

AMAZON. **What is a data lake?** 2021. Disponível em: <https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/>.

AMUNDSEN. **Amundsen**. 2021. Disponível em: <https://www.amundsen.io/>.

APACHE. **Ranger**. 2021. Disponível em: <https://ranger.apache.org/>.

ARMBRUST, M. et al. Delta lake: high-performance acid table storage over cloud object stores. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 13, n. 12, p. 3411–3424, 2020.

BEHESHTI, A. et al. Coredb: A data lake service. Association for Computing Machinery, New York, NY, USA, p. 2451–2454, 2017. Disponível em: <https://doi.org/10.1145/3132847.3133171>.

BISSO, R. et al. Vazamentos de dados: Histórico, impacto socioeconômico e as novas leis de proteção de dados. **Revista Eletrônica Argentina-Brasil de Tecnologias da Informação e da Comunicação**, v. 3, n. 1, 2020.

CELERY. **Celery**. 2021. Disponível em: <https://docs.celeryproject.org/en/stable/index.html>.

CEPH. **Hardware Recommendations**. 2021. Disponível em: <https://docs.ceph.com/en/latest/start/hardware-recommendations/#ram>.

CHA, B. et al. International network performance and security testing based on distributed abyss storage cluster and draft of data lake framework. **Security and Communication Networks**, Hindawi, 2018.

DATABRICKS. **Databricks**. 2021. Disponível em: <https://databricks.com/>.

DEEQU. **Deequ**. 2021. Disponível em: <https://github.com/awslabs/deequ>.

DOCKER. **Docker**. 2021. Disponível em: <https://www.docker.com/>.

DOCKER. **Dockerfile**. 2021. Disponível em: <https://docs.docker.com/engine/reference/builder/>.

DOCKER. **DockerHub**. 2021. Disponível em: <https://hub.docker.com/>.

DREMIO. **Dremio**. 2021. Disponível em: <https://www.dremio.com/>.

EXPECTATIONS, G. **Great Expectations**. 2021. Disponível em: <https://greatexpectations.io/>.

FLEXERA. **State of Cloud Report 2021**. 2021. Disponível em: <https://info.flexera.com/CM-REPORT-State-of-the-Cloud>.

FLOWER. **Flower**. 2021. Disponível em: <https://flower.readthedocs.io/>.

GATLING. **Gatling**. 2021. Disponível em: <https://gatling.io/open-source/>.

GOPALANI, S.; ARORA, R. Comparing apache spark and map reduce with performance analysis using k-means. **International journal of computer applications**, v. 113, n. 1, 2015.

KAFKA, A. **Apache Kafka**. 2021. Disponível em: <https://kafka.apache.org/>.

KEDRO. **Kedro**. 2021. Disponível em: <https://kedro.readthedocs.io/en/stable/>.

KUBERNETES. **Kubernetes**. 2021. Disponível em: <https://kubernetes.io/>.

LANS, R. V. D. **Data Virtualization for business intelligence systems: revolutionizing data integration for data warehouses**. [S.l.]: Elsevier, 2012.

MAEDA, K. Performance evaluation of object serialization libraries in xml, json and binary formats. In: **2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)**. [S.l.: s.n.], 2012. p. 177–182.

MCKINSEY. **Transformações digitais no Brasil: Insights sobre o nível de maturidade digital das empresas no país**. 2019. Disponível em: <https://www.mckinsey.com/br/~/media/mckinsey/locations/southamerica/brazil/ourinsights/transformacoesdigitaisnobrasil/transformacao-digital-no-brasil.pdf>.

MINIO. **MinIO**. 2021. Disponível em: <https://min.io/>.

MIT. **Kerberos**. 2021. Disponível em: <https://web.mit.edu/kerberos/>.

MONGODB. **MongoDB**. 2021. Disponível em: <https://www.mongodb.com/pt-br>.

MORO, G. **Storage Format for Almost-Homogeneous Data Sets**. Dissertação (Mestrado) — ETH Zurich, Zurich, 2020.

MUNSHI, A. A.; MOHAMED, Y. A.-R. I. Data lake lambda architecture for smart grids big data analytics. **IEEE Access**, v. 6, p. 40463–40471, 2018.

ORACLE. **What Is a Data Warehouse?** 2021. Disponível em: <https://www.oracle.com/database/what-is-a-data-warehouse/>.

PANKOV, P.; NIKIFOROV, I.; DROBINTSEV, D. Hardware and software data processing system for research and scientific purposes based on raspberry pi 3 microcomputer. **Proceedings of the Institute for System Programming of the RAS**, v. 32, p. 57–69, 01 2020.

PENG, R. **Kylo Data Lakes Configuration deployed in Public Cloud environments in Single Node Mode**. Tese (Doutorado), 2019. Disponível em: <http://urn.kb.se/resolve?urn=urn:nbn:se:bth-18817>.

PONCE, F.; MÁRQUEZ, G.; ASTUDILLO, H. Migrating from monolithic architecture to microservices: A rapid review. In: IEEE. **2019 38th International Conference of the Chilean Computer Science Society (SCCC)**. [S.l.], 2019. p. 1–7.

PORTAINER. **Portainer**. 2021. Disponível em: <https://www.portainer.io>.

REDIS. **Redis**. 2021. Disponível em: <https://redis.io/>.

SPARK, A. **Apache Spark**. 2021. Disponível em: <https://spark.apache.org/>.

SRINIVASAN, K. et al. An efficient implementation of mobile raspberry pi hadoop clusters for robust and augmented computing performance. **Journal of Information Processing Systems**, Korea Information Processing Society, v. 14, n. 4, p. 989–1009, 2018.

SUPERSET. **Superset**. 2021. Disponível em: <https://superset.apache.org/>.

WARREN, J.; MARZ, N. **Big Data: Principles and best practices of scalable realtime data systems**. [S.l.]: Simon and Schuster, 2015.

APÊNDICE A – Exemplo de Código fonte - Dockerfile

```
1  # especifica uma imagem base
2  FROM raspbian/stretch:latest
3
4  # seleciona o diretorio inicial
5  WORKDIR /home/pi/
6
7  # aplica dependencias necessarias
8  RUN apt update
9  RUN apt upgrade -y
10 RUN wget https://dl.minio.io/server/minio/release/linux-arm/minio && \
11     wget https://dl.minio.io/client/mc/release/linux-arm/mc && \
12     sudo chmod +x /home/pi/minio && \
13     sudo chmod +x /home/pi/mc && \
14     sudo ln -s /home/pi/minio /usr/bin/minio
15     sudo ln -s /home/pi/mc /usr/bin/mc
```

APÊNDICE B – Exemplo de Código fonte - docker-compose.yml

```
1  version: '3'
2  services:
3      dremio-master:
4          build:
5              context: ./
6              dockerfile: ./images/dremio/Dockerfile
7          container_name: dremio-master
8          restart: always
9          volumes:
10             - ./dremio/data:/opt/dremio/data
11             - ./dremio/conf:/opt/dremio/conf
12             #- ./certs:/opt/dremio/certs
13         ports:
14             - "9047:9047"
15             - "31010:31010"
16             - "32010:32010"
17             - "45678:45678"
18         command: >
19             sh -c "bin/dremio-admin upgrade && bin/dremio --
   ↪ config /opt/dremio/conf start-fg"
```

APÊNDICE C – Modelo de Dockerfile para aplicação raspadora de dados - Dockerfile

```
1  # Definicao da imagem base
2  FROM ...
3
4  # Definicao das dependencias
5  ...
6
7  # Definicao das variaveis de ambiente mandatorias
8  ENV MONGODB_HOST "host.docker.internal"
9  ENV MONGODB_PORT 27017
10 ENV MONGODB_DB "staging"
11
12 # Quaisquer operacoes necessarias para a aplicacao
13 ...
```

APÊNDICE D – Modelo para arquivo de definição de raspador de dados - crawler_template.py

```
1  from pymongo import MongoClient
2
3
4  def run(**kwargs):
5      collection = kwargs.get('collection')
6
7      # implementacão do raspador
8      data = ''  # dados raspados
9
10     collection.insert_many(data)
11
12
13 if __name__ == "__main__":
14     params = ""  # input params
15     client = MongoClient(host=os.getenv('MONGODB_HOST'), port=int(
       ↪ os.getenv('MONGODB_PORT')))
16     db = client[os.getenv('MONGODB_DB')]
17     collection = db[os.getenv('MONGODB_COLLECTION')]
18
19     run(params=params, collection=collection)
```

APÊNDICE E – Modelo para arquivo de definição de fonte de dados - sourceMongoDB.json

```
{
  "connectionConfiguration": {
    "ssl": false,
    "host": "host.docker.internal",
    "port": 27017,
    "user": "***",
    "database": "staging",
    "password": "***",
    "auth_source": "admin",
    "replica_set": ""
  }
}
```

APÊNDICE F – Modelo para arquivo de definição de destino de dados - destinationMinIO.json

```
{
  "connectionConfiguration": {
    "format": {
      "format_type": "Parquet",
      "page_size_kb": 1024,
      "block_size_mb": 128,
      "compression_codec": "UNCOMPRESSED",
      "dictionary_encoding": true,
      "max_padding_size_mb": 8,
      "dictionary_page_size_kb": 1024
    },
    "s3_endpoint": "http://HOSTNAME_MAQUINA_MESTRE:PORTA_MINIO",
    "access_key_id": "***",
    "s3_bucket_name": "NOME_BUCKET",
    "s3_bucket_path": "01_bronze",
    "s3_bucket_region": "us-east-1",
    "secret_access_key": "***"
  }
}
```

APÊNDICE G – Modelo para arquivo de definição de conexão do Airbyte - connectorTemplate.json

```
{
  "name": "default",
  "namespaceDefinition": "source",
  "namespaceFormat": "${SOURCE_NAMESPACE}",
  "prefix": "",
  "sourceId": "SOURCE_ID",
  "destinationId": "DESTINATION_ID",
  "operationIds": [],
  "syncCatalog": {
    "streams": [
      {
        "stream": {
          "name": "STREAM_NAME",
          "jsonSchema": {},
          "supportedSyncModes": [
            "full_refresh",
            "incremental"
          ],
          "sourceDefinedCursor": false,
          "defaultCursorField": [],
          "sourceDefinedPrimaryKey": [],
          "namespace": null
        },
        "config": {
          "syncMode": "full_refresh",
          "destinationSyncMode": "append",
```

```
            "primaryKey": [],
            "selected": true
          }
        }
      ]
    },
    "schedule": null,
    "status": "active",
    "resourceRequirements": {
      "cpu_request": null,
      "cpu_limit": null,
      "memory_request": null,
      "memory_limit": null
    }
}
```

APÊNDICE H – Modelo para arquivo de definição de DAG do Airflow - 01_modelo-ingestao.py

```
from datetime import datetime, timedelta
import json
import logging
import os
import requests
import time

from airflow import DAG
from airflow.hooks.base import BaseHook
from airflow.models import Variable
from airflow.operators.python import PythonOperator
from airflow.providers.docker.operators.docker import DockerOperator
from airflow.utils.dates import days_ago
from pymongo import MongoClient

from lib.Airbyte import AirbyteAPI, airbyte_create_connection
from lib.utils import (update_json)

minio_conn = BaseHook.get_connection("minio")
mongodb_conn = BaseHook.get_connection("mongodb")

default_args = {
    'owner': 'admin',
    'depends_on_past': False,
    'email_on_failure': True,
    'email_on_retry': True,
    'start_date': days_ago(1),
    'retries': 0,
    'retry_delay': timedelta(minutes=1)
}
```

```
31
32 dag = DAG(
33     '00_modelo-ingestao',
34     default_args=default_args,
35     description='Track prices from a given Amazon Wishlist'
36                 'via webcrawler, and store it in a MongoDB'
37                 'database.',
38     schedule_interval='0 */4 * * *',
39     catchup=False
40 )
41
42 dag_clean_name = dag.dag_id[3:]
43
44 t1 = DockerOperator(
45     task_id='crawler_container',
46     image='REMOTE_IMAGE_DOCKERHUB',
47     network_mode="host",
48     environment={
49         # "MONGODB_HOST": "host.docker.internal",
50         # "MONGODB_PORT": 27017,
51         "MONGODB_DB": dag_clean_name,
52         "MONGODB_COLLECTION": dag_clean_name
53     },
54     do_xcom_push=False,
55     dag=dag
56 )
57
58
59 def mongodb_cleanup():
60     client = MongoClient(mongodb_conn.host, mongodb_conn.port)
61     client.drop_database(dag_clean_name)
62
```

```
63
64 t2 = PythonOperator(
65     task_id="airbyte_create_connection",
66     python_callable=airbyte_create_connection,
67     dag=dag
68 )
69
70 t3 = PythonOperator(
71     task_id="airbyte_trigger_sync",
72     python_callable=airbyte_trigger_sync,
73     provide_context=True,
74     dag=dag
75 )
76
77 t4 = PythonOperator(
78     task_id="mongodb_cleanup",
79     python_callable=mongodb_cleanup,
80     dag=dag
81 )
82
83 t1 >> t2 >> t3 >> t4
```

APÊNDICE I – Modelo para arquivo de definição de script Pyspark - 02_modelo-pyspark.json

```
import os

from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import DoubleType, TimestampType

spark = SparkSession.builder.getOrCreate()

def extract(format, path):
    return spark.read.format(format).load(path)

def transform(df):
    # remove colunas do airbyte
    df = df.drop(*[column for column in df.columns if "_airbyte" in column]).drop("_id")

    # limpa `createdAt`
    df = df.withColumn('reference_date', F.to_timestamp(F.substring(F.col('createdAt'), 0, 19), "yyyy-MM-dd hh:mm:ss"))
    df = df.drop('createdAt')

    # limpa `availability`
    df = df.withColumn('is_available', F.when(F.col('availability').isNull(), F.lit(True)).otherwise(False))
    df = df.withColumn('availability_date', F.when(F.col('is_available') == F.lit(0),
                                                   F.regexp_extract(F.col('availability'), "\ em\ (.*)\.", 1)))
    df = df.withColumn('availability_date_day', F.split(F.col('availability_date'), "\/", -1)[0])
```

```
25      df = df.withColumn('availability_date_month', F.lower(F.split
        ↪ (F.col('availability_date'), "\/", -1)[1]))
26      df = df.replace(to_replace=meses, subset=['
        ↪ availability_date_month'])
27      df = df.withColumn('availability_date_year', F.split(F.col('
        ↪ availability_date'), "\/", -1)[2])
28      df = df.withColumn('availability_date', F.to_date(
29          F.concat(F.col('availability_date_year'), F.lit('-'), F.
        ↪ col('availability_date_month'), F.lit('-'),
30                   F.col('availability_date_day')), "yyyy-MM-dd"))
31      df = df.drop("availability", 'availability_date_day', '
        ↪ availability_date_month', 'availability_date_year')
32
33      # limpa `review_stars`
34      df = df.withColumn('review_stars_tmp', F.regexp_extract(F.col(
        ↪ 'review_stars'), "^(.{3}).*", 1).cast("double"))
35      df = df.withColumn('review_stars',
36                         F.when(F.col('review_stars_tmp').isNull(), F
        ↪ .lit(-1.0)).otherwise(F.col('review_stars_tmp')))
37      df = df.drop('review_stars_tmp')
38
39      # limpa `sellers_price`
40      df = df.withColumn('has_sellers', F.when(F.col('sellers_price
        ↪ ').isNull(), False).otherwise(True))
41
42      # limpa `delivery_price`
43      df = df.withColumn('has_delivery_tax', F.when(F.col('
        ↪ delivery_price').isNull(), False).otherwise(True))
44      df = df.withColumnRenamed('delivery_price', '
        ↪ delivery_tax_price')
45
46      return df
```

```
47
48 def load(df, path):
49     df.write.format("delta") \
50         .mode("overwrite") \
51         .option("overwriteSchema", "true") \
52         .save(path)
53
54 if __name__ == '__main__':
55     df = extract(os.getenv('source_format', 'parquet'), os.getenv(
       ↪ 'source_path'))
56     df_transformed = transform(df)
57     load(df_transformed, os.getenv('target_path')
```

APÊNDICE J – Modelo para arquivo de definição de DAG do Airflow - 02_modelo-transformacao.py

```
from datetime import datetime, timedelta

from airflow import DAG
from airflow.hooks.base import BaseHook
from airflow.models import Variable
from airflow.providers.apache.spark.operators.spark_submit import
    ↪ SparkSubmitOperator
from airflow.utils.dates import days_ago

airflow_home = Variable.get("AIRFLOW_HOME")
spark_scripts_home = Variable.get("SPARK_SCRIPTS_HOME")
minio_conn = BaseHook.get_connection("minio")

default_args = {
    'owner': 'gabrielmelocomp',
    'depends_on_past': False,
    'email_on_failure': True,
    'email_on_retry': True,
    'start_date': days_ago(2),
    'retries': 1,
    'retry_delay': timedelta(minutes=1)
}

dag = DAG(
    '02_awl-bg',
    default_args=default_args,
    schedule_interval='0 */4 * * *',
    description='',
    catchup=False
)
```

```
30
31 t1 = SparkSubmitOperator(
32     task_id='bronze_to_silver',
33     conn_id='spark_standalone',
34     application=airflow_home + spark_scripts_home + 'pyspark/02_awl
   ↪ -bg.py',
35     packages="org.apache.hadoop:hadoop-aws:3.2.0,com.
   ↪ amazonaws:aws-java-sdk-bundle:.11.980,org.apache.
   ↪ httpcomponents:httpclient:4.5.3,joda-time:joda-time
   ↪ :2.9.9,io.delta:delta-core_2.12:1.0.0",
36     conf={
37         #"spark.deploy.defaultCores": "2",
38         #"spark.deploy.maxExecutorRetries": "2",
39         "spark.worker.cleanup.enabled": "true",
40         "spark.hadoop.fs.s3a.access.key": minio_conn.login,
41         "spark.hadoop.fs.s3a.secret.key": minio_conn.password,
42         "spark.hadoop.fs.s3a.impl": "org.apache.hadoop.fs.s3a
   ↪ .S3AFileSystem",
43         "spark.hadoop.fs.s3a.endpoint": "http{}://{}:{}".
   ↪ format("", minio_conn.host, minio_conn.port),
44         "spark.hadoop.fs.s3a.path.style.access": "true",
45         "spark.sql.extensions": "io.delta.sql.
   ↪ DeltaSparkSessionExtension",
46         "spark.sql.catalog.spark_catalog": "org.apache.spark.
   ↪ sql.delta.catalog.DeltaCatalog"
47     },
48     total_executor_cores=4,
49     num_executors=2,
50     executor_cores=2,
51     executor_memory='512m',
52     driver_memory='512m',
53     name='test-pyspark',
```

```
54      execution_timeout=timedelta(minutes=10),
55      dag=dag
56  )
57
58  t1
```

APÊNDICE K – Modelo de script para criação de conectores no Apache Superset - create_connections_superset.py

```
import json
import requests

headers = {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer <access_token>',
}

data = {
  "configuration_method": "sqlalchemy_form",
  "database_name": "<database>",
  "engine": "dremio",
  "sqlalchemy_uri": "dremio://<user>:<password>@host.docker.
    ↪ internal:31010/dremio"
}

response = requests.post('http://host.docker.internal:8088/api/
    ↪ v1/database/', headers=headers, data=json.dumps(data))
```

APÊNDICE L – Classe principal da biblioteca cliente da API REST do Portainer

```
class PortainerAPI:
    def __init__(self, username, password, host="localhost", port
    ↪ =7999, ssl=False):
        self.host = host
        self.port = port
        self.ssl = ssl
        self.username = username,
        self.password = password,
        self.url = "http{use_ssl}://{host}:{port}/api".format(
            use_ssl="s" if self.ssl else "",
            host=host,
            port=port
        )
        self.base_headers = {
            "Authorization": "Bearer {jwt}".format(
                jwt=self._get_jwt()
            )
        }
        self.stacks = self._get_stacks()

    def _get_jwt(self):
        url = self.url + "/auth"
        data = {
            "username": self.username,
            "password": self.password
        }

        response = requests.post(url=url, data=json.dumps(data))
        try:
            return response.json()["jwt"]
```

```
30          except KeyError:
31              return None
32
33      def _get_stacks(self):
34          url = self.url + "/stacks"
35          response = requests.get(url=url, headers=self.base_headers)
36          stacks = response.json()
37          return {stack["Name"]: stack["Id"] for stack in stacks}
38
39      def stop_stack_by_name(self, stack_name):
40          url = self.url + "/stacks/{id}/stop".format(id=self.stacks
    ↪ [stack_name])
41          response = requests.post(url=url, headers=self.base_headers)
42          return response.json() is not None
43
44      def start_stack_by_name(self, stack_name):
45          url = self.url + "/stacks/{id}/start".format(id=self.
    ↪ stacks[stack_name])
46          response = requests.post(url=url, headers=self.base_headers)
47          return response.json() is not None
```