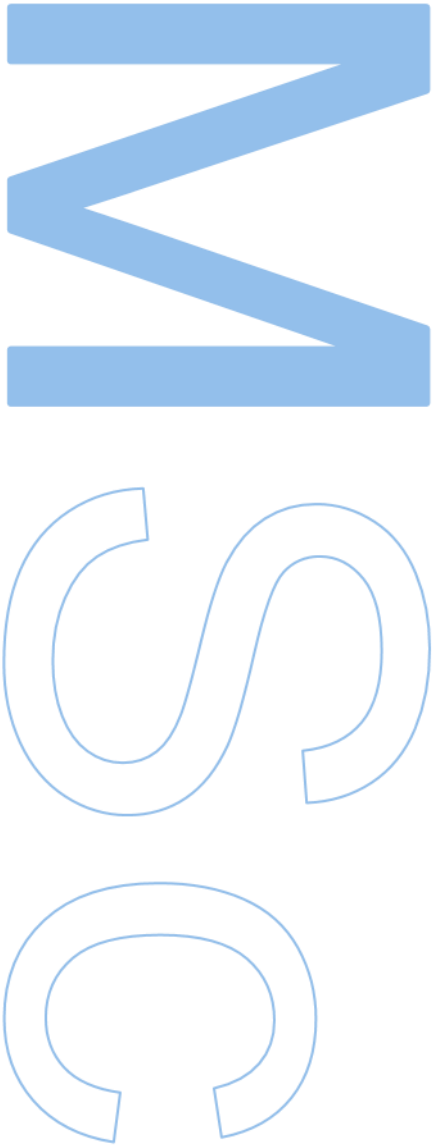
Migração do Sistema REACT para Databricks

João Tiago Mota
Mestrado em Engenharia Matemática
Departamento de Matemática da Faculdade de Ciências
2023

Orientador
Prof. Dra. Inês Dutra, Professora Associado, Faculdade de Ciências

Coorientador
Prof. Dra. Ana Freitas, Professora Associado, Faculdade de Economia




U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Agradecimentos

Queria começar por expressar a minha profunda gratidão às minhas orientadoras académicas, à professora Dra. Inês Dutra e à professora Dra. Ana Freitas. O vosso apoio, paciência e dedicação ao longo deste projeto significou imenso para mim e para o desenvolvimento deste trabalho. Agradeço imenso por cada momento despendido, por cada conselho, por cada reunião, por cada e-mail e pela imensa ajuda que me proporcionaram. Obrigado!

Gostaria também de expressar a minha gratidão aos meus orientadores na Sonae, Daniela Rocha e Tiago Vilela. A vossa confiança e a oportunidade que me deram significaram imenso para mim e foram fundamentais para o meu crescimento. Agradeço por acreditarem em mim, pela responsabilidade que me atribuíram e pelo espaço que me deram para aprender de forma autónoma. Obrigado!

Queria também agradecer aos meus amigos, que de forma direta ou indireta me ajudaram na escrita desta tese e que estiveram sempre ao meu lado, ao longo desta tese, do meu percurso académico e da minha vida em geral. Também queria fazer um agradecimento especial ao meu amigo Manuel Romão por me ter ajudado com a minha escrita disléxica ao longo deste relatório. Obrigado!

Por fim, e não por último queria agradecer à minha família, à minha mãe, Elisa Mota, ao meu pai, Manuel Mota e à minha irmã, Ana Mota, pelo apoio incondicional que me deram ao longo da minha vida toda, pelo amor, carinho e por tudo que fizeram por mim. Tudo que sou hoje é devido a estas três pessoas. Muito obrigado!

UNIVERSIDADE DO PORTO

Resumo

Faculdade de Ciências da Universidade do Porto
Departamento de Matemática

Mestrado em Engenharia Matemática

Migração do Sistema REACT para Databricks

por João Tiago MOTA

Este relatório descreve o trabalho desenvolvido durante um estágio no âmbito do Mestrado em Engenharia Matemática na Faculdade de Ciências da Universidade do Porto, realizado na empresa Sonae, no centro de excelência de *Business Intelligence*.

As empresas de retalho, numa incessante busca por competitividade, veem-se na necessidade de oferecer uma gama diversificada de produtos de alta qualidade a preços competitivos. Neste contexto, a tomada de decisão com base em dados torna-se cada vez mais importante. No entanto, o grande volume de dados e a rapidez com que devem ser processados apresentam desafios para infraestruturas computacionais tradicionais. Estas muitas vezes não conseguem evoluir a par com as exigências tecnológicas atuais, gerando lacunas de otimização no processamento e análise de dados.

Neste cenário, surge a relevância de arquiteturas e tecnologias mais modernas, tal como o Databricks, que é uma plataforma baseada no Apache Spark e operado em *cloud*. Oferece soluções robustas para o tratamento de grande volume de dados.

É desta necessidade de arquiteturas e tecnologias mais modernas que este relatório surge. Durante este relatório será explorado o estudo e a realização de uma migração de um algoritmo (REACT) da plaforma Cloudera Machine Learning (CML) para Databricks. O REACT foi desenvolvido em 2019 para abordar desafios cruciais da equipa de desenvolvimento comercial não alimentar da Sonae MC. Este algoritmo nasceu da necessidade de gerir problemas associados a produtos descontinuados e produtos com níveis elevados de *stock*. Permitindo uma gestão mais eficaz do *stock* de produtos, centralizando-se numa abordagem pro ativa do ciclo de vida dos produtos através da monitorização ativa e de estratégias de baixa de preço.

UNIVERSIDADE DO PORTO

Abstract

Faculdade de Ciências da Universidade do Porto
Departamento de Matemática

MSc. Mathematical Engineering

Migration of the REACT System to Databricks

by João Tiago MOTA

This report describes the work carried out during an intership as part of the Master´s in Mathematical Engineering at the Faculty of Sciences of the University of Porto at the company Sonae, in the Business Intelligence center of excellence.

In an incessant search for competitiveness, retail companies need to offer a diverse range of high-quality products at competitive prices. In this context, decision-making based on data is becoming increasingly important. However, the sheer volume of data and the speed with which it must be processed present challenges for traditional computing infrastructures, which often fail to evolve in line with current technological requirements, creating optimization gaps in data processing and analysis.

In this scenario, the relevance of more modern architectures and technologies emerges, such as Databricks, which is a platform based on Apache Spark and operated in cloud. It offers robust solutions for handling large volumes of data.

It is from this need for more modern architectures and technologies that this report arises. This report explores the study and implementation of a migration of an algorithm (REACT) from Cloudera Machine Learning (CML) platform to Databricks. REACT was developed in 2019 to address crucial challenges faced by Sonae MC's non-food team. This algorithm was born out of the need to manage problems associated with discontinued products and products with high levels of stock. It enables more effective management of product *stock*, focusing on a proactive approach to the product lifecycle through active monitoring and price reduction strategies.

Conteúdo

Lista de Figuras

Code Listings

Glossário

Soane MC	Sonae Modelo Continente
API	Application Programming Interface
CML	Cloudera Machine Learning
DB	Databricks
ADF	Azure Data Factory
GFS	Google File System
MP	MapReduce
HDFS	Hadoop Distributed File System
Hadoop YARN	Yet Another Resource Negotiator
RM	Resource Manager
MPP	Massive Parallel Processing
DF	DataFrame
RDD	Resilient Distributed Dataset
DSL	Domain Specific Language
ETL	Extract, Transform, Load
ML	Machine Learning
PVP	Preço de Venda ao Público
SKU	Stock Keeping Unit

Capítulo 1

Introdução

Num cenário de constante evolução tecnológica, a otimização e eficiência dos algoritmos desempenha um papel de extrema importância para as organizações. Na busca por manter a sua competitividade, muitas empresas procuram plataformas capazes de oferecer melhorias significativas no desempenho e eficácia, o que torna a migração entre plataformas uma decisão estratégica. É nesse contexto que este trabalho surge, na necessidade de migrar um algoritmo da plataforma Cloudera Machine Learning para a plataforma Databricks. Ao longo deste capítulo, serão exploradas as motivações e objetivos deste trabalho, juntamente com a contextualização da Sonae e da definição da estrutura deste dissertação.

1.1 Motivação e Objetivos

As empresas de retalho estão constantemente à procura de maneiras para se manterem competitivas, com o intuito de atrair o maior número possível de clientes. Uma das principais estratégias é oferecer uma ampla gama de produtos de alta qualidade, a preços competitivos. No entanto, para tomar decisões informadas sobre quais produtos oferecer, a que preços, como melhorar os seus negócios, assegurar rentabilidade e reduzir desperdício de *stock*, estas empresas precisam de ter acesso a um grande volume de dados e, mais importante, conseguir processar e analisar estes dados de uma forma rápida e eficaz.

Contudo, muitas das infraestruturas computacionais associadas a estas empresas não conseguem acompanhar a rápida evolução tecnológica na área do tratamento de grandes quantidades de dados, deixando lacunas de otimização.

No entanto, com o uso de arquiteturas e tecnologias mais modernas, estas empresas podem ultrapassar estas lacunas de otimização e obter *insights* valiosos sobre os seus dados. Uma das tecnologias mais modernas para o processamento de dados é o Databricks, que é uma plataforma de análise de dados em *cloud* baseada no Apache Spark. Oferece uma série de recursos que podem ajudar estas empresas a processar grandes quantidades de dados de forma rápida e eficiente, incluindo processamento em tempo real, *machine learning* e análise preditiva.

É nessa vertente que este trabalho surge, com o objetivo de migrar e otimizar do processo REACT. Este processo foi desenvolvido para auxiliar na tomada de decisão de baixa de preço durante o fim do ciclo de vida dos artigos, garantindo o escoamento eficiente destes produtos. Isto, como referido anteriormente, é crucial, uma vez que, caso estes produtos não sejam escoados dentro do prazo, ocuparão espaço valioso no armazém, resultando em desperdício de recursos e redução da rentabilidade da empresa. Atualmente, este processo encontra-se na plataforma Cloudera e apresenta deficiências de escalabilidade e de desempenho. A Sonae MC tomou a decisão estratégica de migrar os seus processos da plataforma Cloudera para a platorma Databricks, com o objetivo de potenciar os benefícios económicos e operacionais que esta oferece.

O objetivo deste trabalho consiste em:

1. Realizar a migração do processo REACT da plataforma Cloudera Machine Learning para a plataforma Databricks, assegurando que o processo atual em CML permaneça inalterado. Durante a migração, será priorizada a transição do código de Pandas para PySpark, com o intuito de aproveitar a capacidade de processamento oferecida pela plataforma Databricks;

2. Identificar e otimizar partes do código que possam apresentar redundâncias ou ineficiências, com o objetivo de melhorar a eficácia e eficiência do processo.

1.2 Caracterização da Sonae MC

A Sonae, Sociedade Nacional de Estratificados, foi criada em 1959 pelo empresário Afonso Pinto de Magalhães, tendo como origem trabalhos nas áreas das madeiras processadas, mais concretamente, na produção de painéis laminados decorativos de alta-pressão.

Na década de 80, a Sonae começou a expandir internacionalmente para diferentes áreas de negócios, começando a construir as fundações do grupo atual. Em 1985 dá-se a abertura do primeiro hipermercado em território português: o Continente, em Matosinhos. Esta abertura tem por base a criação da Sonae MC *, fazendo do retalho alimentar um dos principais negócios do grupo Sonae (Sonae, s.d.-a).

Ao longo dos anos, a MC foi aumentando o seu portefólio de marcas: Bagga (cafetaria, pastelaria e padaria), Continente (hipermercado), Continente Auto (lavagem e/ou aspiração de viaturas), Continente Bom Dia (supermercados de conveniência), Continente Labs (Continente sem caixa), Continente Modelo (antigos hipermercados Modelo), Continente Online (comércio eletrónico do Continente), Dr. Well´s (clínicas de medicina dentária, estética e capilar), elergone energia (empresa focada na otimização da fatura energética), Go Natural (restaurantes e supermercados de alimentação saudável), Home Story (loja de mobiliário e decoração), Meu Super (lojas de proximidade em formato de franquia), Note! (livraria, papelaria e presentes), Seguros Continente (vários tipos de seguros), SKINERI-ETM (marca de cosméticos), Well´s (saúde, bem-estar e beleza), ZU (loja de animais) e Washy (lavandarias self-service) (Sonae, s.d.-b). Atualmente a MC possui mais de 1300 lojas e um volume de negócios superior a 5100M€, mais de quatro milhões de famílias como clientes e mais de trinta e seis mil colaboradores.

Em 2020, a Sonae encontrava-se presente em 62 países. Desta presença destaca-se o retalho, serviços financeiros, imobiliário, telecomunicações, entre outros (Sonae, s.d.-c).

1.3 Estruturação do Documento

Ao longo deste trabalho, será abordado o desafio de otimizar o processo REACT, desenvolvido para tomar decisões estratégicas sobre a baixa de preço durante o ciclo de vida dos artigos. O objetivo principal deste trabalho é migrar e otimizar este processo, reconhecendo a importância de assegurar a rentabilidade das operações e reduzir o desperdício de recursos, tendo origem num plano de modernização das arquiteturas da Sonae MC.

O presente documento está estruturado em seis capítulos cuidadosamente organizados, com o propósito de assegurar uma compreensão profunda e coesa dos tópicos abordados. O primeiro capítulo dedica-se à contextualização do problema em estudo, fornecendo uma análise histórica da Sonae e ao objetivo deste trabalho. O segundo capítulo apresenta conceitos essenciais ao trabalho, como por exemplo, conceitos fundamentais

*Sonae MC - Sonae Modelo Continente

sobre *cloud*, plataformas de *cloud*, fatores de desempenho e boas práticas para migração de algoritmos. No terceiro capítulo proporciona-se uma contextualização abrangente sobre o processo REACT, destacando as suas limitações, os estudos realizados e delineando um plano de melhorias. No quarto capítulo aborda-se detalhadamente o processo de migração, descrevendo as alterações implementadas durante este procedimento. No quinto capítulo apresenta-se os resultados obtidos, acompanhados de uma análise de desempenho. O documento conclui com o sexto capítulo, onde se encerra este trabalho por meio da apresentação de conclusões e lições aprendidas, além de esboçar algumas perspetivas para trabalhos futuros.

Capítulo 2

Background e Estado da Arte

Neste capítulo procede-se à contextualização dos conceitos mais importantes relevantes a esta migração. Inicialmente, é apresentado o termo *Big Data* e a sua história, seguido da descrição das arquiteturas nas quais o processo REACT está inserido, nomeadamente Hadoop e Cloudera Machine Learning e onde estes falham comparativamente a arquiteturas mais modernas. Posteriormente, são abordados os conceitos relacionados com a plataforma a ser migrada, especificamente Spark e Databricks.

2.1 Big Data

No decorrer da evolução tecnológica e do aumento da capacidade de armazenamento de dados, a disponibilidade de dados para processamento e análise tem crescido extraordinariamente. No entanto, esse avanço tecnológico também trouxe consigo desafios técnicos significativos. Antes da invenção dos computadores, os cientistas realizavam os cálculos à mão. Com o aparecimento dos computadores, o processamento de dados passou a realizar-se de uma forma mais rápida, sendo geralmente possível lidar com a quantidade de dados existentes na altura. A necessidade de armazenar e processar grandes quantidades de dados foi aumentando ao longo dos anos, sendo esta necessidade continuamente satisfeita pelo rápido avanço da capacidade de armazenamento e dos *hardwares* dos computadores. Porém, com a explosão da internet, os dados começaram a crescer de uma forma exponencial e os *hardwares* não conseguiram acompanhar este crescimento (Cukier & Mayer-Schoenberger, 2013).

Não se sabe com certeza quem usou o termo *Big Data* pela primeira vez, contudo muitos creditam a origem a John Mashey, um cientista de computadores que trabalhou na

Silicon Graphics nos anos 90. Ele usou o termo para descrever as bases de dados que estavam a ficar grandes e complexas para as ferramentas de gestão de dados existentes na altura. Outra utilização do termo, que gerou um crescimento no seu uso, foi de Roger Mougalas, em 2005, que trabalhava na *O'Reilly Media*, que definiu *Big Data* como conjunto de dados que eram demasiado grandes para serem trabalhados com ferramentas de gestão de dados tradicionais (Gupta & Rani, 2019).

O termo *Big Data* refere-se a conjuntos de dados tão grandes e complexos que ultrapassam a capacidade de processamento e análise dos sistemas tradicionais em tempo tolerável. Não há um limite fixo de *gigabyte's* para definir *Big Data*, mas sim uma questão relativa à capacidade do computador em lidar com o volume, velocidade e variedade dos dados. Por exemplo, para computadores pessoais modernos com RAM entre 8 a 32 GB uma quantidade de dados de 10-20 GB pode começar a ser desafiadora, especialmente durante operações intensivas, ou seja, operações que requerem um alto consumo de recursos computacionais, tais como, transformação de dados e modelagem de *machine learning*. Já para as empresas com recursos de computação com maior quantidade de armazenamento, por exemplo, um *warehouse* ou servidores em *cloud*, a escala dos dados pode ser significativamente maior, podendo lidar com centenas de *gigabyte's* ou até mesmo *terabyte's* de informações complexas. Em geral, o conceito de *Big Data* é melhor definido pelo impacto que estes conjuntos de dados têm nas operações e na tomada de decisões, em vez de uma métrica de tamanho específico.

Em fevereiro de 2001, Doug Laney publicou um artigo (Laney, 2001) que se baseia nas suas experiências com *Big Data*, tendo referido que *Big Data* poderia ser descrita com três V´s, estes são:

- ***Volume***, em português, Volume, refere-se à grande quantidade de dados gerados que ultrapassam a capacidade das ferramentas tradicionais de armazenamento e processamento;
- ***Velocity***, em português, Velocidade, refere-se à velocidade com que os dados são criados;
- ***Variety***, em português, Variedade, quer dizer que *Big Data* vem de várias formas diferentes. Pode vir em forma estruturada (tabelas de dados com linhas e colunas), de forma semi-estruturada (como por exemplo os ficheiros JSON) ou de forma não

estruturada (áudio, imagem, vídeos). Cada um destes formatos de dados apresenta diferentes problemas no processamento.

Genericamente, estes são os três V´s originais que definem *Big Data*. Contudo, têm vindo a criar-se mais V´s, tais como *Value* (referindo-se ao valor que os dados podem trazer para uma organização), *Validity* (que está relacionado com a validade dos dados. Dados validados são dados que estão corretos e adequados para a utilização pretendida) e *Variability* (referindo-se à inconsistência dos dados ao longo do tempo), entre muitos outros. Apesar da criação destes, os três originais são os mais aceites (Cukier & Mayer-Schoenberger, 2013).

O Google foi das primeiras empresas que identificou este crescimento exponencial de dados, tendo um papel pioneiro no desenvolvimento de soluções para resolver este problema, chegando mais tarde a publicar artigos que partilhavam as soluções. O primeiro artigo, chamado de "Google File System", abordava como resolver o problema de armazenar e de gerir de uma forma eficiente os dados em grande escala, tendo sido publicado em 2003 (Ghemawat et al., 2003). Um ano mais tarde, publicaram o segundo, intitulado de "MapReduce programming model", que apresenta o modelo de programação MapReduce, que é uma abstração que permite o processamento distribuído e paralelo de grandes volumes de dados em servidores. De uma forma geral, aborda o processamento e transformação dos dados (Dean & Ghemawat, 2004).

Estes trabalhos foram fundamentais para a comunidade de código aberto e tiveram um impacto significativo na criação de muitos softwares, especialmente o Hadoop, que mais tarde, devido às suas limitações com o modelo MapReduce, levou à criação do Apache Spark, uma tecnologia mais avançada e versátil para processamento de *Big Data*.

2.2 Hadoop

O Hadoop é uma plataforma de software de código aberto, criada por Doug Cutting e por Mike Cafarella em outubro de 2003. Foi desenvolvida para armazenar e processar grandes quantidades de dados de forma distribuída em vários *clusters*, inicialmente usando *hardware* convencional, mas posteriormente evoluindo para suportar *hardware's* mais sofisticado. A sua criação foi inspirada pelos conhecimentos fornecidos pelo Google para se desenvolver. A plataforma é constituída por três módulos principais (Kala Karun & Chitharanjan, 2013):

1. Hadoop Distributed File System;
2. Hadoop MapReduce;
3. Hadoop Yarn.

Além destas componentes centrais, o ecossistema Hadoop, também inclui tecnologias como Hive e Impala, que oferecem funcionalidades SQL-on-Hadoop. O Hive fornece uma interface SQL para consultar e processar dados armazenados no Hadoop, enquanto que o Impala permite efetuar consultas interativas em tempo real em grandes volumes de dados.

No entanto, o Hadoop possui algumas limitações notáveis, especialmente em relação ao processamento em tempo real e à complexidade de programação do modelo MapReduce. Sendo neste contexto que surge o Apache Spark, uma estrutura de processamento de dados rápida e flexível que supera muitas dessas limitações.

2.2.1 Hadoop Distributed File System

O Hadoop baseou-se no conhecimento adquirido com o GFS para implementar o HDFS. O HDFS é um sistema de ficheiros distribuídos projetado para operar em vários *clusters* de computadores, permitindo o armazenamento distribuído de grandes volumes de dados. Em termos de funcionamento, o HDFS precisa de dois nós (*nodes*): o nó mestre (*namenode*) e os nós trabalhadores (*datanodes*).

O nó mestre, como o nome diz, atua como o mestre. Contém todos os metadados relacionados à estrutura de diretórios e ficheiros, mantendo registo de qual nó trabalhador contém partes específicas dos ficheiros, conforme ilustrado na Figura 2.1. Além disso, também é o responsável por gerir as operações de ficheiros, como: abrir, fechar, mudar o nome, acrescentar, entre outros.

Por outro lado, os nós trabalhadores, como o nome diz, são os trabalhadores do mestre: estes armazenam os dados em blocos e são encarregados de operações de leitura e escrita solicitadas pelo nó mestre.

O HDFs oferece alto rendimento e é otimizado para processamento de dados em grande escala. Contudo, apresenta um acesso de dados com alta latência. Isto deve-se ao facto de ter sido especificamente desenhado para processamento em massa, e não necessariamente para tarefas interativas ou que precisem de resposta imediata.

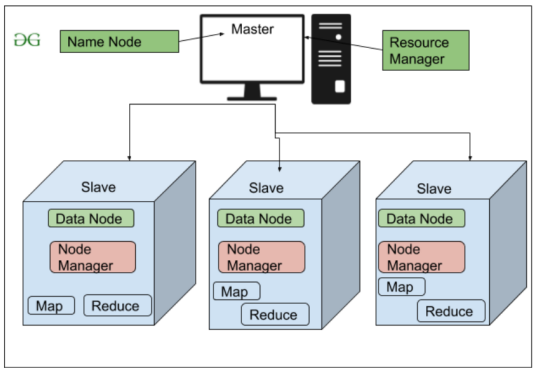


FIGURA 2.1: Nós Mestre e Trabalhador (Arquitetura Hadoop) (Merceedi & Sabry, 2021)

2.2.2 MapReduce

O Hadoop MapReduce foi desenvolvido com base no do Google MapReduce. Este modelo é baseado no método de divisão e de conquista. Um grande problema é dividido em múltiplos subproblemas que são simples e pequenos o suficiente para serem resolvidos diretamente num só computador. Estes subproblemas são processados de forma independente e paralela. Finalmente, quando os cálculos estão todos resolvidos, a solução destes subproblemas é agregada para dar a solução do problema inicial.

A execução deste modelo apoia-se, principalmente, em duas funções: Map e Reduce. A primeira, distribui os dados em todos os diferentes clusters que existem e gera os resultados numa forma de pares <chave; valor> (<key; value>). Entre a função Map e Reduce, podem existir funções intermédias, tais como, *filter, group by, sort, join, left, outer join, right outer join*. A função Reduce finaliza as operações intermédias e aplica operações como *count, reduce by key, group by key, first, union, cross*, e fornece o resultado num par <key, value> que é de fácil compreensão pela parte do utilizador. Como se pode ver, na Figura 2.2, ilustra-se um exemplo de como contar o número de ocorrências das palavras que aparecem em dois ficheiros, aplicando as funções Map e Reduce (Merceedi & Sabry, 2021).

2.2.3 Hadoop YARN

O Hadoop começou como uma *framework* de código aberto que implementava a metodologia MapReduce, com o objetivo de processar e guardar grandes quantidade de dados. Contudo, com o passar dos anos, o número de utilizadores a usar o Hadoop também foi crescendo e, devido a este crescimento, começaram a existir manuseamentos errados do

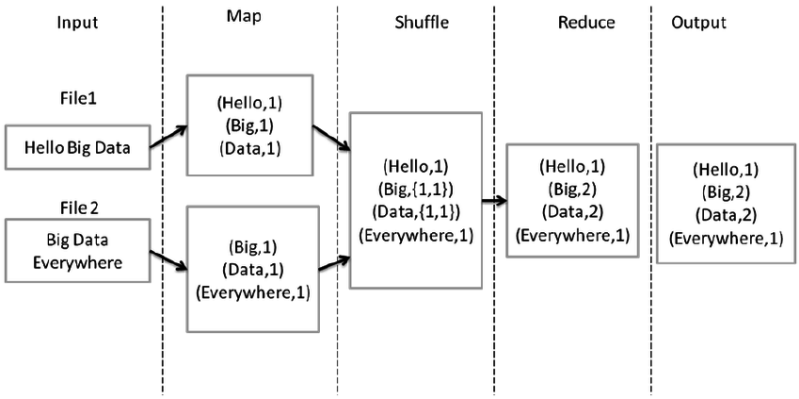


FIGURA 2.2: Exemplo de contagem de palavras usando o método de MapReduce («Different phases of map reduce in word count example», s.d.)

Hadoop. Os utilizadores tentavam utilizar o MapReduce para realizar a gestão dos recursos (*Resource Management*) e os seus agendamentos (*Job Scheduling*), mesmo o MapReduce não estando desenhado para estas ações.

Foi então que se desenvolveu o Hadoop YARN, que foi lançado no Hadoop 2 para ultrapassar e melhorar as limitações do MapReduce. O MapReduce começou então a ser popularmente denominado como MapReduce 1 (o que se encontrava diponível na versão 1 e anterior do Hadoop), para se distinguir do MapReduce 2 que utiliza a implementação do Hadoop YARN (White, 2012).

A ideia principal do Hadoop YARN é permitir a separação das funcionalidades de gestão de recursos e dos agendamentos das tarefas em seções separadas, o que possibilitou otimizar o processamento em paralelo, sendo estas enunciadas seguidamente («Apache Hadoop Yarn», s.d.) (Vavilapalli et al., 2013):

1. *Resource Manager*: Aceita as submissões de tarefas, agendando os trabalhos e alocando os recursos (CPU e memória) necessários. Tem dois grandes componentes:

   (a) *Scheduler*: É aqui que cai a responsabilidade de alocar os recursos para as aplicações em execução. Não executa nenhuma monitorização.

   (b) *Application Manager*: Local de coordenação das execuções do *Application Master* no cluster, isto é, é aqui onde cai a responsabilidade de começar um *Application Master*, da sua monitorização e da sua reiniciação em diferentes nós em caso de falha.

2. *NodeManager (NM)*: Funcionalidade onde se lança e se monitoriza os *compute containers* e as utilizações dos diferentes recursos (CPU, memória, disco e *network*), reportando ao *Resource Manager/Scheduler*

3. *Application Master (AM)*: Permite a coordenação das tarefas. O AM negoceia os diferentes recursos ao RM e trabalha com o NM para executar e monitorizar as tarefas.

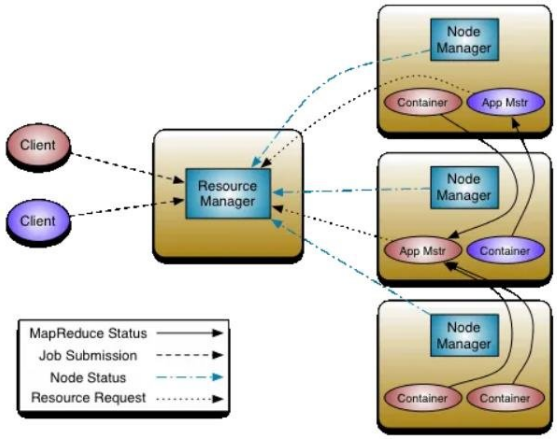


FIGURA 2.3: Arquitetura Hadoop YARN («Apache Hadoop Yarn», s.d.)

2.2.4 *SQL-on-Hadoop*

Como referido, o Hadoop é um software de armazenamento e processamento tanto de dados estruturados, como não estruturados. Como tal, tornou-se cada vez mais necessário o desenvolvimento de ferramentas de análise para dados armazenados no HDFS. Utilizou-se o SQL, visto que esta é a linguagem mais comum de se usar para armazenamento e manipulação de dados.

O *SQL-on-Hadoop* é uma combinação de duas ferramentes poderosas que combinam a escabilidade e a flexibilidade do Hadoop com o poder de consulta do SQL. Permitiu às empresas armazenar grandes quantidades de dados no Hadoop utilizando consultas do tipo SQL para analisar esses dados, permitindo aos programadores analisar grandes quantidades de dados de forma mais rápida e eficiente.

Como o SQL foi inicialmente desenvolvido para ser trabalhado com bases de dados relacionais, foi preciso modificá-lo para se conseguir trabalhar com ele em Hadoop. Foi, então, necessário utilizar três técnicas distintas para se trabalhar com SQL em Hadoop:

1. Conectores que traduzem SQL para um formato MapReduce;

2. Sistemas *push-down* que realizam *batch-oriented* MapReduce e executam SQL dentro dos clusters do Hadoop;

3. Sistemas que transferem SQL entre clusters de MapReduce-HDFS ou HDFS clusters.

A introdução de SQL no Hadoop abriu várias portas trazendo vários benefícios, tais como: **produtividade**, visto que a maioria dos analistas já sabia programar em SQL e não houve uma necessidade de formações ou de contratações extras; mais **flexibilidade**, dado que os sistemas tradicionais apenas funcionavam para dados estruturados. Contudo, com o HDFS sendo o repositório central (i.e., *Data Lake*), dados de várias fontes, sendo estruturados ou não, já podem ser analisados com mais facilidade, entre outros benefícios (Maposa & Sethi, 2018).

Foram desenvolvidos vários *softwares* que suportavam análise do tipo SQL em dados armazenados no HDFS. O primeiro destes foi o *Apache Hive* ou só *Hive*. Não sendo suficiente, este programa não satisfez o que era pretendido, levando à criação de muitos mais *softwares* de SQL em Hadoop, como por exemplo o Impala (Qin et al., 2017).

2.2.4.1 Apache Hive

Como referido anteriormente, o *Apache Hive* foi o primeiro *SQL-on-Hadoop* a ser desenvolvido. É uma estrutura de armazém de dados de código aberto que possui a capacidade de gerir, consultar, processar e analisar um grande volume de dados guardados em HDFS. Foi desenvolvido com o propósito de simplificar o acesso ao *MapReduce*, utilizando a linguagem *HiveQL*, que é uma linguagem declarativa semelhante à linguagem utilizada em SQL, que converte *queries* em tarefas do MapReduce na plataforma Hadoop (Qin et al., 2017).

Não obstante, as operações realizadas em *Hive*, por norma, continuaram lentas devido ao *MapReduce*, tendo sido desenvolvidas outras opções (Qin et al., 2017).

O Apache Hive tem sido bastante usado em várias empresas para gerir e processar grandes quantidade de dados, como por exemplo: Yahoo, Facebook, Spotify, entre outros (Qin et al., 2017).

2.2.4.2 Impala

O Impala, desenvolvido em C++ e em Java, é um *software* de código-aberto de SQL de processamento em paralelo em massa, ou em inglês, *Massive Parallel Processing* (MPP). Foi

desenhado para um desempenho em tempo real, com pouca latência, com alta capacidade de processamento em paralelo e com a ideia de combinar o SQL com a escabilidade e flexibilidade do Hadoop (Qin et al., 2017).

A redução da latência é explicada pela utilização do *MapReduce* ou pela leitura de dados de forma remota. A parte da alta capacidade de processamento em paralelo é explicada pelo facto de que o Impala implementa uma arquitetura distribuída que pode correr em centenas de máquinas que já existam no cluster do Hadoop, sendo esta responsável pela execução de consultas (Qin et al., 2017).

2.2.4.3 Apache Hive VS Impala

Na tabela 2.1, apresenta-se um estudo comparativo entre o Apache Hive e o Impala.

TABELA 2.1: Comparação de características entre Apache Hive e Impala (Qin et al., 2017)

Características	Hive	Impala
Processamento em Paralelo	Suporta	Suporta
Fault Tolerance	Vai depender do *fault tolerance* do Hadoop	Durante o processo de *query´s* não existe, se ocorrer um erro durante a execução, uma mensagem de erro será enviada
Scheduling	O agendamento de tarefas depende da estratégia de agendamento do Hadoop	O agendamento é realizado pelo utilizador
Latência	Como foi construido usando o *MapReduce* do Hadoop, possui uma maior latência	O Impala foi construído a pensar na velocidade e, por esta razão, possui uma latência muito baixa
Complexidade das *Queries*	O *Hive* foi construído para aguentar longas *queries*	O *Impala* foi construído para aguentar *queries* mais pequenas em bases de dados maiores

2.2.5 Onde o Hadoop falha e onde o Spark surge

A ineficiência do Map Reduce para determinadas tarefas, como, por exemplo, quando existem múltiplas iterações de processamento, ou a baixa velocidade de processamento, é devido ao facto que o Hadoop lê e grava os dados no disco durante cada etapa do processo aumentando assim a latência. Este problema e muitos outros fizeram surgir a necessidade de criar outras *frameworks* para combater estes problemas. Algumas destas são o Dryad, Tez e o Apache Spark (Zaharia et al., 2010).

2.3 Apache Spark

O Apache Spark é uma ferramenta de código aberto destinada ao processamento de grandes volumes de dados de forma paralela e distribuída. A plataforma suporta a criação de APIs em quatro linguagens de programação: Scala, Java, Python e R. Surgiu em

2009, por Matei Zaharia no âmbito de um projeto para a unidade curricular AMPLab (Algorithms, Machines and People Lab) na Universidade da Califórnia. Surgiu como um esforço para simplificar e otimizar o modelo de programação MapReduce (Zaman, 2023).

O modelo MapReduce exige que os utilizadores implementem as funções Map e Reduce, que resulta numa estrutura de fluxo de dados linear. A função Map lê os dados de um disco físico e mapeia estes dados. Já a função Reduce agrega os resultados produzidos pela função Map e, em seguida, grava o *output* no disco. O Spark, por outro lado, introduziu a capacidade de iterar sobre os dados, algo que era inviável com o modelo MapReduce tradicional. Esta nova capacidade de iterar sobre os dados abriu portas para novos métodos de processamento de *big data*, como: machine learning e processamento de grafos. Esses recursos são disponibilizados pelas bibliotecas do Spark: MLlib e GraphX (Zaman, 2023).

Uma das principais inovações do Spark foi a introdução de *Resilient Distributed Dataset* (RDDs), uma estrutura de dados imutável e distribuída que pode ser processada em paralelo. Os RDDs proporcionam tolerância a falhas e permitem um processamento de dados em memória altamente eficiente, resultando num desempenho superior ao do modelo MapReduce (Salloum et al., 2016).

2.3.1 Resilient Distributed Dataset - RDD

Um *Resilient Distributed Dataset*, conhecido como RDD, é definido como uma coleção imutável e repartida de elementos que é tolerante a falhas e que é capaz de operar em paralelo (Spark, s.d.-c).

Devido à natureza técnica do tema, será explicado em detalhe as propriedades fundamentais dos RDDs:

1. ***Immutable***: uma vez criado, o seu conteúdo não pode ser alterado;
2. ***Fault-tolerant or resilient***: há cópia de segurança dos dados que permitem a sua recuperação em caso de erro ou falha;
3. ***Partitioned or distributed***: os dados são distribuídos e repartidos entre os diferentes nós do cluster;
4. ***Operated on in parallel***: os dados são processados simultaneamente em paralelo.

Existem várias formas diferentes de criar uma RDD (Zaharia et al., 2010) (Spark, s.d.-c):

1. ***Parallelizing***: é um método para criar uma RDD a partir de uma coleção já existente, presente no driver, por exemplo: um array ou uma lista:

```
rdd = spark.sparkContext.parallelize([11,12,13,14,15])
rdd.collect()
# Output:
[11,12,13,14,15]
```

CODE LISTING 2.1: Exemplo de criação de RDDs através de Parallelizing

2. ***Referencing***: uma RDD pode ser criada diretamente a partir de ficheiros armazenados em sistemas de ficheiros como HDFS, por exemplo. O Spark também suporta vários formatos, como CSV, Parquet e outros:

```
rdd_txt = spark.sparkContext.textFile("file_name.txt", 10)
```

CODE LISTING 2.2: Exemplo de criação de RDDs através de Referencing

3. ***A partir de outras RDDs***: transformações em RDDs existentes, produzem novas RDDs:

```
rdd2 = rdd.map(lambda x: x * 2)
```

CODE LISTING 2.3: Exemplo de criação de RDDs através de outros RDDs

Após a criação de RDDs, podem ser aplicados dois tipos principais de operações: *Transformations* e *Actions*.

1. ***Transformations***: As transformações criam novos *datasets* (pois os RDDs são imutáveis) através da aplicação de operações ao *dataset* original. Estas transformações são designadas de *preguiçosas* ou *lazy evaluated*. Isto significa que elas só são executadas quando uma ação é invocada. Até esse momento, elas permanecem numa fila de espera, durante a qual o Spark constrói e otimiza um plano de execução. Este comportamento ajuda a reduzir o uso excessivo de recursos nas máquinas.

   Por exemplo, ***filter(func)*** é uma transformação que permite manter ou remover dados de forma condicional. Se quisermos remover os valores *None* da seguinte RDD, podemos usar a função filtro:

```
# input RDD [11,12,None,14,15]
rdd.filter(lambda x: x is not None)
# output RDD [11,12,14,15]
```

CODE LISTING 2.4: Exemplo da transformação filter

A transformação ***map(func)*** aplica uma operação a cada elemento da RDD. Por exemplo, se quisermos somar um a cada elemento da seguinte RDD, faz-se o seguinte:

```
# input RDD [11,12,13,14,15]
rdd.map(lambda x: x+1)
# output RDD [12,13,14,15,16]
```

CODE LISTING 2.5: Exemplo da transformação map

Mais exemplos de transformações são: *flatMap(func)*, *reduceByKey(func)*, *union(otherRDD)*, *join(otherRDD)*, *Cartesian*, *groupByKey()*, entre outras (Spark, s.d.-c).

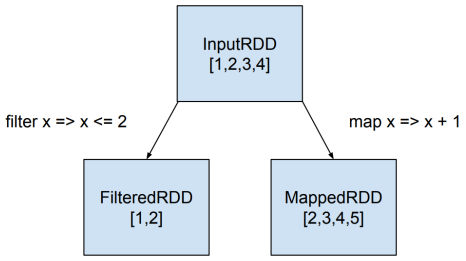


FIGURA 2.4: Exemplo de criação de novas RDDs usando as transformações filter(func) e map(func)

2. ***Actions***: Enquanto as transformações criam novos *dataset* a partir de *dataset* existentes sem a realização imedata de computações, as ações efetivamente realizam essas computações nesses *dataset* e retornam valores.

   Por exemplo, ***reduce(func)*** é uma ação que vai agregar os elementos do RDD usando a função dada.

```
rdd = spark.sparkContext.parallelize([11,12,13,14,15]) #Criar a RDD
rdd.reduce(lambda x,y: x+y)
# Output:
65
```

CODE LISTING 2.6: Exemplo de uma ação

A ação ***collect()*** retoma todos os valores do conjunto de dados como um *array*. É muito útil quando se usa umas transformações *filter(func)* ou outras que retornam um conjunto de dados pequeno.

```
01 | rdd = spark.sparkContext.parallelize([11,12,13,14,15]) #Criar a RDD
02 | rdd.map(lambda x : x+1)
03 | #Ao correr a linha anterior, o output seria:
04 | PythonRDD[5] at RDD at PythonRDD.scala:53
05 | #Por isso, tem de se usar a acao .collect() para se visualizar a
       transformacao:
06 | rdd.map(lambda x: x+1).collect()
07 | #Output:
08 | [12, 13, 14, 15, 16]
09 | #Outro exemplo, seria contar o numero de elementos:
10 | rdd.count()
11 | #Output:
12 | 5
```

CODE LISTING 2.7: Exemplo de uma ação

Outros exemplos de ações são *take(n)*, *first()*, *count()*, entre outros (Spark, s.d.-c).

Contudo, nas versões mais recentes do Spark os RDDs estão a ser substituídos por *DataFrames* e *Datasets*, devido às suas melhores otimizações e APIs mais fáceis de usar. Ambos representam avanços significativos no processamento de dados distribuídos, proporcionando otimizações de desempenho.

Os *Dataframes* foram introduzidos na versão 1.3.0 do Spark. Conceptualmente são equivalentes a uma tabela numa base de dados relacionais ou a um *data frame* no R/Python. Foram criados para facilitar a realização de operações comuns, como seleção, filtragem, agregação e junção de dados de maneira declarativa, sem a necessidade de escrever código de baixo nível. Foram otimizados para um desempenho melhor do que os RDDs aproveitando características do motor Spark (R. Xin et al., 2015).

Por outro lado, os *Datasets* foram introduzidas na versão 1.6.0 do Spark, estes combinam características dos RDDs e dos *DataFrames*, oferecendo uma abstração de dados mais robusta e poderosa (Armbrust et al., 2016).

2.3.2 Ecossistema Spark

O ecossistema Spark é dividido em duas partes:

1. A parte inferior é designada como Spark Core;
2. A parte superior é constituída por DSL´s (domain-specific language), bibliotecas e API´s.

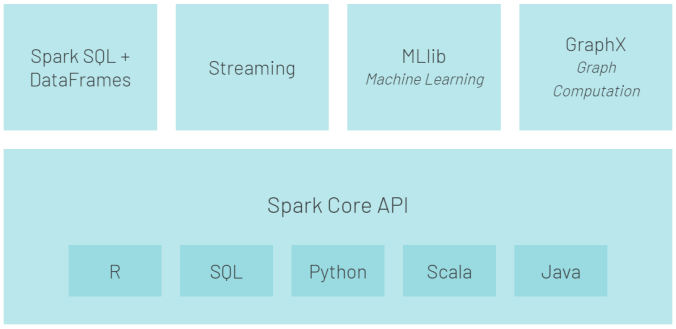


FIGURA 2.5: Ecossistema Spark (Damji & Lee, s.d.)

2.3.2.1 Spark Core

É a fundação do ecossistema Spark. Esta camada contém componentes essenciais que garantem a execução distribuída, o agendamento de tarefas, a recuperação de falhas, interação com sistemas de armazenamento e muito mais. O Spark Core contém API´s para se trabalhar com os RDDs, que são o principal conceito de Spark.

2.3.2.2 Spark SQL

Este módulo permite o processamento de dados estruturados usando a interface SQL. Os dados podem ser acedidos por ficheiros JSON, Parquet, Hive Tables, entre outros. Permite, também a manipulação de dados através de SQL e HQL (Apache Hive Query, a versão SQL desenvolvida pela Apache).

2.3.2.3 Spark Streaming

Spark Streaming é um módulo para aplicações em que é necessário analisar um conjunto de dados que chegam em tempo real, fornecendo a abstração Discretized Streams (DStreams) que simplifica a tarefa de trabalhar com múltiplos data streams e que suporta dados de sites como Kafka, Flume, Twitter, entre outros.

2.3.2.4 MLlib

MLlib é uma biblioteca de Machine Learning do Spark. Contém várias funcionalidades de ML, como por exemplo: classificação, regressão, clustering, redução de dimensionalidade e filtragem colaborativa. Também oferece funcionalidades como avaliação de modelos e importação dos dados. Foi construída para funcionar em clusters de servidores, tornando assim o processamento mais rápido (Spark, s.d.-b).

2.3.2.5 GraphX

GraphX é um componente que permite a manipulação de grafos e computação paralela dos mesmos. Esta biblioteca estende-se dos APIs de Spark aos RDDs e usa a abstração de Directed Multigraph com propriedades em cada vértice e arestas para representação de dados. Possui operadores de manipulação de grafos, tais como subgraph e joinVertices, e possui também algoritmos de grafos, tais como PageRank e Triangle Count (R. S. Xin et al., 2014).

2.3.3 Arquitetura Spark

O Spark segue um modelo de comunicação mestre/trabalhador de forma similar à do Hadoop, onde o mestre é o *driver*. As aplicações são desenhadas de forma a que tenham um nó coordenador central, ou driver, e diversos nós trabalhadores, ou cluster, aos quais o nó mestre distribui os trabalhos.

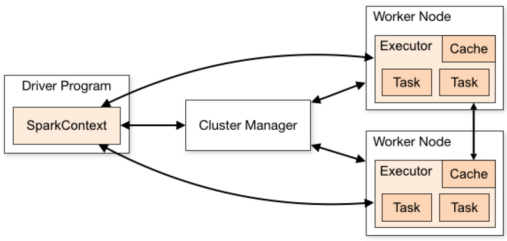


FIGURA 2.6: Arquitetura Apache Spark (Spark, s.d.-a)

2.3.3.1 Driver

O *driver* é o cerébro de uma aplicação Spark e é responsável por três tarefas: manter a informação sobre a aplicação Spark, responder ao *input* do utilizador e analisar, distribuir e agendar as tarefas nos executores. Usualmente o Spark processa os dados em porções

de 64MB, 128 MB ou 256MB sendo estes distribuídos nos diferentes nós trabalhadores. Pode-se correr o driver fora do cluster, usando o modo cliente ou pelo cluster usando o modo cluster. Idealmente, deve-se usar o driver numa máquina com boa conexão à internet, visto que o driver necessita de estar em contacto constante com os executores. Se iniciarmos o Spark ou fizermos uma submissão do mesmo num cluster, o *SparkContext* será criado pelo Driver dentro do nó mestre. O *SparkContext* é como a porta para as funcionalidades Spark (Databricks, 2020).

2.3.3.2 Executador

Os executores do Spark são os processos que executam as tarefas atribuídas pelo *driver*. Basicamente, os executores tem as seguintes responsabilidades: pegar nas tarefas atribuídas pelo *driver* e executá-las, reportando o seu estado, isto é, se houve uma falha ou se foi bem sucedido e os resultados das tais execuções. Os executores são criados nos nós trabalhadores a pedido do nó mestre (Databricks, 2020).

2.3.3.3 Cluster Manager

O *Cluster Manager* é uma plataforma onde se pode correr o Spark. Muito resumidamente, quando o objeto *SparkContext* (no driver) é criado, ele liga-se ao *Cluster Manager* para negociar o número de executores necessários.

O Spark suporta três tipos de *cluster manager* (Databricks, 2020):

1. **Standalone**: Gestor de *cluster* nativo do Spark, é ideal para *clusters* de menor dimensão;

2. **Hadoop YARN**: gestor de recursos de Hadoop, que permite ao Spark e às aplicações do Hadoop partilharem o mesmo *cluster*;

3. **Apache Mesos**: é um sistema de gestão de *clusters* de propósito geral, escalável e flexível, adequado para grandes *clusters* e diversas aplicações distribuídas.

2.3.4 Modo de Execução

Os modos de execução ajudam a determinar onde é que o programa corre. Existem dois modos de execução:

2.3.4.1 Client Mode

Neste modo o driver corre localmente no computador do cliente, no entanto, o driver ainda se liga ao cluster manager e inicia a ligação com os executores no cluster. Contudo, após desligar o computador, o driver morre e, por consequência disso, os executores também morrem. Este é, por isso mesmo, um modo mais indicado para um trabalho mais interativo, como por exemplo trabalhar com o spark-shell e com *notebook*s, não tanto para um trabalho de longa duração. Este é o modo *default* do Spark.

2.3.4.2 Cluster Mode

Este modo é desenhado para submeter a nossa aplicação ao cluster e deixá-la correr. Neste modo, ao contrário do anterior, tudo corre no cluster. Por esse motivo, pode-se desligar o computador e o driver não é impactado, sendo um modo mais indicado para trabalhos de mais longa duração. É a maneira mais comum de correr um programa Spark.

2.3.5 Resumo

O Spark é uma ferramenta que permite processar grande quantidade de dados de forma paralela e distribuída, tornando-se fundamental para quem trabalha com *Big Data*. Um exemplo conhecido de uma empresa que utiliza o Spark é o *TripAdvisor* (Palmucci, s.d.).

Ao longo dos anos foram-se criando plataformas para análise de *big data* utilizando Spark. Alguns exemplos são:

1. Databricks
2. Delta Lake
3. Apache Mesos
4. Rumble (Apache)

2.3.6 MapReduce VS Spark

No artigo (Zaharia et al., 2012), realiza-se uma experiência para avaliar o desempenho e a escalabilidade do Spark em relação ao Hadoop.

Foram realizadas duas experiências: aplicações iterativas de *machine learning* e PageRank, um algoritmo que mede a importância das páginas na internet com base na quantidade e qualidade dos links de entrada que apontam para essas mesmas páginas.

No caso das aplicações interativas de *machine learning*, implementaram-se dois modelos: regressão logística e K-Means. Executou-se ambos os algoritmos para 10 iterações num conjunto de dados de 100 GB. No final, concluiu-se que o Spark foi 25.3x mais rápido do que no Hadoop na regressão logística e 1.9x mais rápido no K-Means (que é um algoritmo que requer mais recursos computacionais comparativamente à regressão logística). Esta diferença deve-se ao facto de que os RDDs mantém os dados intermediários em memória, enquanto que o Hadoop MapReduce lê e grava os dados no disco em cada iteração.

No caso do algoritmo PageRank, utilizou-se 54GB de conteúdo da Wikipedia. Executaram-se 10 iterações do algoritmo, o equivalente a aproximadamente 4 milhões de artigos. Os resultados desta experiência são interessantes: no caso de armazenamento na memória o Spark obteve resultados 2.4x melhores que o Hadoop. Os autores também observaram que ao aumentar o tamanho do cluster de 30 nós para 60 nós, o desempenho do Spark escalou quase linearmente. Isto sugere que os RDDs são uma abstração eficiente e escalável para algoritmos iterativos como o PageRank.

2.3.7 PySpark

O Spark foi originalmente desenvolvido em Scala, uma linguagem de programação funcional e orientada a objetos. Isto implicava que se uma pessoa quisesse programar em Spark teria de aprender Scala. No entanto, para reduzir esta barreira foi lançado o PySpark. O PySpark é uma biblioteca Python para Spark que atua como uma interface de alto nível*, minimizando a barreira de trabalhar com Spark, permitindo aos utilizadores utilizarem as suas capacidades, nomeadamente manipulação de RDDs de forma distribuída e em Python (Databricks, 2022). Contudo, é importante notar que nem todas as funcionalidades do Spark estão disponíveis em PySpark, um destes motivos é devido às diferenças das duas linguagens de programação.

*Interface de Alto Nível: é uma interface que é criada para ser fácil de usar e enteder pelos utilizadores, independente do seu nível de experiência. Neste caso, o PySpark facilta o trabalho com o Spark ao abstrair muitas das complexidades e detalhes técnicos do Spark, permitindo aos utilizadores os recursos do Spark sem terem de entender completamente os detalhes internos do Spark.

2.4 Pandas VS PySpark

Pandas e PySpark são duas bibliotecas populares usadas para manipulação e análise de dados em Python. Enquanto que Pandas é uma ferramenta eficiente para manipulação de dados para um conjunto pequeno e médio de dados, o PySpark é usado principalmente para conjuntos de dados grandes, ou *big data*. Ambos têm vantagens e desvantagens. Na Tabela 2.2, encontra-se algumas comparações de características entre estas duas bibliotecas.

TABELA 2.2: Comparação de características entre Pandas e PySpark (Zaman, 2023)

Características	Pandas	PySpark
Escalabilidade	É escalável para pequenos e médios conjuntos de dados, tornando-se lento para um conjunto de dados que excede a memória de uma única máquina	É altamente escalável para grandes conjuntos de dados, pois distribui o processamento de dados num cluster, podendo ser ineficiente para um conjunto pequeno de dados
Velocidade	É rápido para pequenas e médias operações de dados, tornando-se lento para operações que envolvem grandes conjuntos de dados	É mais rápido em conjuntos de dados muito grandes, devido ao seu processamento distribuído, tornando-se lento para pequenos conjuntos de dados
Execução	Segue uma estratégia de execução *eager*, ou imediata, i.e., as operações são executadas assim que são chamadas	É executada de forma *lazy*, ou preguiçosa, i.e., o PySpark regista as operações e só as executa quando realmente for necessário o resultado
Mutabilidade	Os *data frame* são mutáveis, i.e., permite operações como adição ou remoção de linhas	Os *data frames* são imutáveis, i.e., uma vez criado, o *data-frame* não pode ser alterado
Tolerância a falhas*	Se um erro ocorrer, o processo inteiro pode falhar e a menos que haja algum mecanismo externo implementado, não haverá recuperação do ponto de falha	Como faz parte do ecossistema Spark, é tolerante a falhas, usando um sistema de linhagem para saber como o *data frame* foi construído e ocorrendo uma falha ele consegue recriar a parte perdida
Exemplo	pdDataFrame.count() retorna o número de observações sem NA por cada coluna	pyspark.sql.DataFrame.count() retoma o número de linhas presentes no *data frame*

Apesar das várias diferenças entre Pandas e PySpark também existem alguns aspetos

semelhantes, como por exemplo, o suporte para diferentes formatos de dados, incluindo CSV, Excel, JSON, Parquet, entre outros. Tanto o Pandas como o PySpark permitem a execução de consultas SQL nos seus *data frames* e permitem a realização de múltiplas operações sobre os dados, tais como filtragem, seleção, agregação, entre outras.

Para a maior parte das diferenças entre Pandas e PySpark, existem vantagens e desvantagens, o que significa que a escolha entre utilizar Pandas ou PySpark não é empírica, mas sim dependente de cada caso de uso específico. Conforme mencionado na Tabela 2.2, o PySpark consegue ser mais lento nas operações sobre dados comparativamente ao Pandas para conjuntos de dimensão reduzida. Isto deve-se ao facto de que a utilização de PySpark requer a inicialização de um cluster Spark e a distribuição dos dados, o que pode ser ineficiente para conjunto de dados de menor dimensão (Stančin & Jović, 2019).

2.5 Cloudera

A Cloudera é uma empresa que fornece uma plataforma de *software* baseada no Apache Hadoop para a gestão e análise dados, permitindo às organizações realizar análises de dados em grande escala, utilizando recursos de processamento distribuído.

O produto principal é a Plataforma de Dados Cloudera (CDP), que é uma plataforma que permite às empresas gerirem, analisarem e extraírem informação valiosa a partir tanto de dados não estruturados quanto de dados estruturados. Oferece também uma variedade de serviços e capacidades, incluindo processamento de dados, armazenamento de dados, *machine learning* e análise de dados em tempo real.

2.5.1 Cloudera Machine Learning

O Cloudera Machine Learning é um dos serviços oferecidos pelo CDP. É uma plataforma de *machine learning* e ciência de dados em ambiente de produção que oferece aos cientistas de dados a capacidade de construir, treinar, testar e implementar modelos num ambiente seguro e escalável.

2.6 Databricks

O software Databricks foi lançado em 2013 e, à semelhança do Apache Spark, foi desenvolvido na unidade curricular AMPLab por Ali Ghodsi, Andy Konwinski, Ion Stoica, Patrick Wendell, Reynold Xin, Matei Zaharia e Arsalan Tavakoli. É uma plataforma de

código aberto em nuvem, criada com o objetivo de simplificar os desafios operacionais associados ao Spark e transformá-lo num *framework* de computação distribuída de código aberto recorrendo à linguagem Scala. É uma plataforma de análises unificada*, tornando-a uma ferramenta muito mais viável para empresas (Palmucci, 2015) que utilizam Apache Spark, permitindo gerir os *cluster* do Spark na nuvem. Atualmente, permite aos cientistas de dados, engenheiros de dados e analistas de negócios trabalharem na mesma plataforma, sem se terem de preocupar com a manutenção de infraestruturas.

Em vez de utilizar um Spark de código aberto (OSS) †, o Databricks oferece uma versão otimizada do Spark chamada de *Databricks Runtime* (DBR), que é um conjunto de artefactos de *software*‡ que são executados nos *clusters* das máquinas do Databricks. Para além de incluir otimizações e melhorias para o Spark, mas também inclui vários componentes e atualizações que permitem melhorar substancialmente a usabilidade, o desempenho e a segurança. Oferecendo integrações com outras tecnologias, tais como o Apache Hadoop. Uma das suas principais características é a sua capacidade de escalabilidade, permitindo aumentar ou diminuir a capacidade do processamento de acordo com a necessidade.

A plataforma apresenta *notebook*'s *web* colaborativos que permitem escrever e executar códigos em várias linguagens de programação, como Python, R, Scala e SQL, implementando a sua execução nos clusters acima mencionados. Como o Databricks foi integrado com o Apache Spark, este é capaz de processar e analisar dados em paralelo, tornando-se assim uma ferramenta poderosa para o manuseamento de *Big Data*. Também proporciona um ambiente unificado para o desenvolvimento de ETL's, permitindo aos negócios gerirem as suas *pipelines* de dados numa única plataforma, reduzindo a complexidade e melhorando assim a eficiência. Além disso, oferece um amplo conjunto de serviços para engenheiros de dados, como gestão do *data lake*, pipelines de ETL e autonomização do *data warehouse*.

*Plataforma de análises unificadas é uma plataforma que permite integrar diversos recursos e tecnologias num sítio só, como por exemplo, armazenamento de dados, processamento de dados, ferramentas de visualização, aprendizagem de máquina, inteligência artificial

†um software de código aberto ou em inglês *Open-source software* (OSS) é um software de computador que é lançado com uma licença na qual o detentor dos direitos de autor permite os direitos de uso, de estudar e distrubuir o software e o código-fonte a todos e com qualquer prepósito

‡Um conjunto de artefactos de software é um conjunto de componentes que são necessários para o funcionamento do software, como por exemplo, código-fonte, bibliotecas, arquivos de conFiguração, documentação, entre outros

2.6.1 Data Lake vs Data Lakehouse vs Data Warehouse

A plataforma Databricks foi desenvolvida a partir da ideia de *data lakes*. Estes *lagos de dados* são repositórios centrais de dados, que os armazena na sua forma original, isto é, armazena tanto dados estruturados (tabelas de bases de dados, folhas do Excel), como semiestruturados (ficheiros XML, páginas WEB) e não estruturados (imagens, ficheiros de áudio, tweets). Os dados são armazenados em zonas ou camadas diferentes, dependendo da sua origem e tipo. Esta estrutura permite que diferentes tipos de utilizadores utilizem os dados de acordo com as suas necessidades específicas. Uma vantagem dos *delta lakes* é a flexibilidade referida no armazenamento de dados, contudo algumas desvantagens estão relacionadas com a falta de estruturação dos dados e à necessidade de processamento adicional dos dados para sua análise (Databricks, s.d.-f) (Databricks, s.d.-c).

Os conceitos de *data lake* e *data warehouse* são semelhantes, no que toca à sua natureza, ambos armazenam e processam dados. Contudo os *data warehouse* têm uma natureza relacional, armazenando apenas dados processados e estruturados de acordo com um esquema previamente definido, sendo estes organizados em ficheiros e pastas. Geralmente possui recursos de gestão de dados, como limpeza e extração/carga/transformação (ETL) (Databricks, s.d.-f).

O Databricks é uma plataforma que leva os conceitos de *data lake* e *data warehouse* mais longe, implementando o conceito inovador de *data lakehouse*. Este conceito representa uma evolução nas capacidades de gestão e análise dos dados, combinando o armazenamento flexível de dados não estruturados de um *data lake* com as avançadas ferramentas de gestão de um *data warehouse* (Databricks, s.d.-f).

Esta nova abordagem permite às empresas armazenarem grandes volumes de dados brutos quer estes sejam estruturados, semiestruturados ou mesmo não estruturados. Esta plataforma também oferece um ambiente altamente escalável e de alto desempenho, permitindo aos utilizadores dos dados uma maior flexibilidade para explorar os dados, sem ter de impor uma estrutura previamente.

Ao mesmo tempo, oferece várias ferramentas avançadas para a gestão e análise dos dados, tais como a otimização de consultas, o processamento em paralelo distribuído, integração com linguagens de programação e bibliotecas de análise de dados (Databricks, s.d.-c).

2.6.2 Delta Lake

No contexto do Databricks, o conceito de *Data Lakehouse* é implementado via *Delta Lake* que é uma camada de armazenamento otimizada e de gestão de dados, que foi desenvolvida para superar os desafios encontrados em plataformas de *big data* (Databricks, s.d.-g).

O conceito de *Delta Lake* é baseado no formato *parquet*, que é um formato colunar otimizado para leitura e compressão eficiente de dados, oferecendo uma estrutura de armazenamento eficaz, permitindo o acesso rápido aos dados, especialmente quando apenas uma certa parte deles precisa de ser lida. Para além disso, o *parquet* suporta compressão avançada, o que reduz o espaço necessário para o armazenamento de dados (Databricks, s.d.-g).

O *delta lake* estende o conceito de *parquet* adicionando recursos avançados, tal como o conceito das transações ACID (Atomicidade, Consistência, Isolamento, Durabilidade). Estas transações garantem que as operações de gravação e de leitura sejam consistentes e duráveis, mesmo em situações de falha, isto é especialmente relevante para aplicações críticas e análises em tempo real, onde a integridade dos dados é fundamental. Além disso, também fornece o controlo da versão para os dados armazenados, permitindo aceder e restaurar versões anteriores dos dados com muita facilidade (Databricks, s.d.-g).

Ao criar uma tabela em Databricks, por defeito, ela é uma Tabela Delta. Estas tabelas são uma forma estruturada de organizar e gerir os dados dentro do *delta lake*, aproveitando todas as vantagens desta camada.

2.6.3 Tabelas Delta

As tabelas delta, ou em inglês *Delta Tables* são uma forma de armazenamento de dados em Databricks que oferece vários benefícios comparativamente a outros formatos de armazenamento tais como *Parquet* ou Hive, sendo uma componente-chave para a gestão eficaz dos dados. Estas tabelas foram desenvolvidas para trazer confiança e melhor desempenho para *big data*.

As tabelas delta oferecem uma série de propriedades que as tornam ideais para trabalhar com um grande volume de dados. Algumas destas propriedades são (Databricks, s.d.-g) (Databricks, s.d.-e) (Databricks, s.d.-b):

1. **Operações ACID:** as tabelas delta suportam transações ACID, o que torna as transações atômicas, consistentes, isoladas e duráveis;

2. **Histórico de versões:** as tabelas delta mantêm um registo de todas as alterações feitas ao conjunto de dados;

3. **Compactação:** as tabelas delta compactam automaticamente o conjunto de dados, melhorando o desempenho das consultas;

4. **Escalabilidade:** as tabelas delta foram projetadas para serem escaláveis, o que significa que podem ser usadas para armazenar e processar uma grande quantidade de dados

No contexto de processamento de grandes volumes de dados, a otimização do desempenho é uma preocupação fundamental. Para lidar com isto, o Databricks implementou várias funcionalidades de manipulação de Tabelas Delta. Entre estas, destacam-se a funcionalidade de *cache* e de *broadcast*, que melhoram a eficiência do processamento de dados:

1. ***Cache:*** é uma técnica que implica o armazenamento temporário de dados (o databricks permite o armazenamento dos dados tanto na memória, como no disco ou nos dois, por exemplo) frequentemente consultados, para que estes sejam consultados de forma mais rápida em consultas futuras. Por exemplo, suponha-se que se está a trabalhar com um *data frame* sobre vendas de produtos que possui milhões de linhas. Se várias operações, como filtragem, agrupamento e cálcular estatísticas, forem necessárias, o Spark lerá o *data frame* a cada operação. Ao usar a técnica do *cache*, o Spark alocará o *data frame* na memória após a primeira leitura, tornando as seguintes mais rápidas por serem realizadas diretamente da memória, que é uma operação significantemente mais rápida do que a leitura a partir do disco (Databricks, s.d.-d).

2. ***Broadcast:*** é uma técnica usada para otimizar a eficiência de operações de união/-junção no Spark. Esta operação distribui um *data frame* pequeno a todos os nós do cluster, permitindo que estes realizem operações localmente, sem necessidade de comunicar com outros nós. Por exemplo, existe um grande *data frame* de vendas de produtos e um pequeno *data frame* de informação de produtos. Se o objetivo for unir estes dois *data frames*, ou seja, adicionar a informação dos produtos a cada venda, tal tarefa, poderia ser dispendiosa em termos de tempos e recursos, dado que implicaria a transferência de dados entre os diferentes nós do cluster. Através da técnica do *broadcast*, o Spark distribui uma cópia do pequeno *data frame* para todos os nós do cluster, permitindo que cada nó obtenha a informação do produto localmente, evitando assim a necessidade de comunicação com outros nós (Databricks, s.d.-a).

2.6.4 Utilização de Widgets no Databricks

Os *widgets* em Databricks permitem interagir dinamicamente com os *notebook*s em Databricks, facilitando a exploração de dados de maneira interativa, personalizando a exibição de dados com base nas preferências do utilizador (Databricks, s.d.-h).

Existem quatro tipos principais de *widgets* no Databricks (Databricks, s.d.-h):

1. ***Widgets* de texto**: Permite aos utilizadores inserir uma *string* de texto;
2. ***Widgets* de *dropdown***: Permite aos utilizadores selecionar uma opção de uma lista *dropdown*.
3. ***Widgets* de *combobox***: Similar ao widget *dropdown*, mas também permite a entrada de texto pelo utilizador.
4. ***Widgets* de *multiselect***: Permite aos utilizadores selecionar várias opções de uma lista.

Depois de um *widget* ser criado, ele aparece na parte superior do *notebook*, como se observa na Figura 2.7:

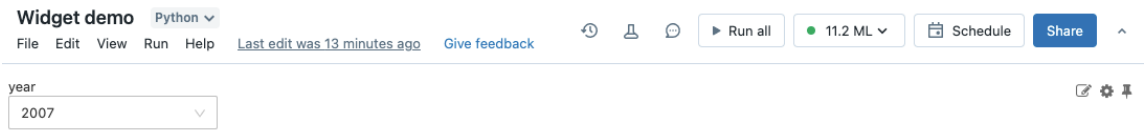


FIGURA 2.7: Exemplo de um *widget* num *notebook* em Databricks (Databricks, s.d.-h)

Os *widgets* podem ser úteis em diversas situações, como por exemplo, a utilização de um *widget* de texto para especificar o nome da tabela que os utilizadores querem criar, ou um *widget dropdown* para selecionar entre diferentes anos para filtrar um conjunto de dados. Isso facilita a reutilização de *notebook*s para análises diferentes, tornando-os mais interativos e fáceis de usar.

2.7 Cloudera VS Databricks

Com base nos dois subcapítulos previamente apresentados, é possível destacar algumas vantagens do Databricks em relação à plataforma de dados Cloudera (CDP) para a migração de dados:

1. Unificação de processamentos de dados e tecnologia de IA: O Databricks é uma plataforma unificada que integra processamentos de dados e tecnologia de Inteligência Artificial. Isso permite que as empresas aproveitem recursos avançados de análise de dados e IA num único ambiente. A integração nativa dessas tecnologias pode facilitar a migração e a implementação de soluções de análise de dados avançadas.

2. Gestão de *cluster* do Spark na *cloud*: O Databricks permite gerenciar *clusters* do Apache Spark na *cloud*. Isso oferece flexibilidade e escalabilidade, pois os recursos computacionais podem ser facilmente aprovisionados e ajustados de acordo com as necessidades da migração de dados. Essa capacidade de gestão simplificada pode agilizar o processo de migração.

3. *Databricks Runtime* otimizado: O Databricks oferece uma versão otimizada do Apache Spark chamada *Databricks Runtime* (DBR). Essa versão é projetada para melhor desempenho e eficiência, o que pode acelerar o processamento de dados durante a migração. A otimização do Spark pelo Databricks pode resultar em tempos de resposta mais rápidos e menor consumo de recursos.

4. Conjunto abrangente de serviços para engenheiros de dados: O Databricks fornece um amplo conjunto de serviços para engenheiros de dados, incluindo gestão do *data lake*, *ETL pipelines* e autonomização do *data warehouse*. Esses recursos podem simplificar e agilizar o processo de migração, fornecendo ferramentas integradas para várias etapas do *data workflow*.

Embora a plataforma de dados Cloudera (CDP) também ofereça recursos como autoatendimento, *multicloud* e segurança, o Databricks destaca-se pela sua integração nativa de tecnologias avançadas, gestão simplificada de *clusters* Spark na *cloud*, otimização do Spark com o *Databricks Runtime* e conjunto abrangente de serviços para engenheiros de dados. Essas características podem tornar o Databricks uma opção mais viável para a migração de dados em questão, especialmente porque a SONAE procura uma solução que unifique processamentos de dados, tecnologia de IA e gestão simplificada na *cloud*.

2.8 Microsoft Azure

Microsoft Azure é uma plataforma e infraestrutura de computação em nuvem criada em 2010 com o objetivo de criar, implementar e gerir aplicações e serviços através de

uma rede global de *data centers* geridos pela Microsoft. Fornece vários serviços incluindo (Microsoft, s.d.-c):

1. *Storage*: O Azure fornece serviços de armazenamento tanto para dados estruturados como para dados não estruturados, tais como Azure Blob Storage e Azure Files;
2. *Analytics*: O Azure fornece várias ferramentas e serviços para analisar dados, incluindo Azure Data Lake e Azure HDInsight (um serviço gerido pelo Apache Hadoop);
3. *Machine Learning*: O Azure fornece serviços de ML tais como Azure Machine Learning e Azure Machine Learning Studio.

Adicionalmente, uma componente significativa do ecossistema Microsoft Azure é o Azure Databricks, que é um serviço de análise baseado em Apache Spark otimizado para a plataforma Azure, que se integra com outras componentes, como por exemplo o Azure Data Factory.

2.8.1 Azure Data Factory

O Azure Data Factory é uma plataforma de serviços de integração de dados baseados na nuvem que permite criar, agendar e orquestrar *pipelines*, ou fluxos de dados que podem ser ingeridos de várias fontes. Com esta plataforma pode-se construir processos complexos de ETL, facilitados por uma interface gráfica intuitiva. Esta interface permite uma visualização do fluxo de dados, tornando mais simples a tarefa de manipular e transformar os dados entre várias fontes e serviços de computação, como Hadoop, Azure Databricks e Azure SQL Database. Resumidamente, em Azure Data Factory pode-se efetuar as seguintes tarefas:

1. Criar e agendar *pipelines* ou fluxos de dados;
2. Processar ou transformar esses dados através de serviços de computação, tais como Hadoop, Spark, Azure Data Lake Analytics, entre outros;
3. Publicar dados de saída em sítios como Azure Synapse Analytics, para o negócio os consumir.

Com o Azure Data Factory é possível repartir os dados pelo tempo, ou seja, pode-se produzir dados de saída de hora a hora, diariamente, semanalmente, etc (Microsoft, s.d.-b).

2.8.2 Azure Data Lake Storage Gen2

O Azure Data Lake Storage Gen2 (ADLS Gen 2) é uma solução de armazenamento em grande escala da Microsoft para análises de *big data*. Contém várias funcionalidades, incluindo acesso compatível com Hadoop (projetado principalmente para trabalhar com todas as arquiteturas que utilizam HDFS como camada de acesso de dados), notável escalabilidade e custos e desempenhos otimizados, entre outras características (Microsoft, s.d.-a).

2.9 Migração de Algoritmos Entre Plataformas

A migração de um algoritmo/processo para uma nova plataforma pode ser uma tarefa complexa que requer várias considerações, mas também apresenta uma oportunidade para melhorar o algoritmo inicial. O objetivo final é garantir que o algoritmo/processo funcione corretamente na nova plataforma sem perda de desempenho ou qualidade.

2.9.1 Análise Inicial das Plataformas

Antes de iniciar a migração, é crucial avaliar as diferenças e semelhanças entre as duas plataformas. Isto inclui um estudo comparativo das linguagens de programação, sistemas operacionais, estrutura de hardware, bibliotecas, entre outros. É fundamental entender as diferenças e potenciais restrições da nova plataforma e como elas podem afetam a migração (Forbes, 2019).

2.9.2 Estudo da Situação Atual

Antes de avançar com a migração para a nova plataforma e após a análise inicial das plataformas e de já saber as possíveis restrições, é fundamental estudar o algoritmo ou o processo atual na plataforma original. Este estudo deve abordar a estrutura geral do código, identificar partes críticas para o desempenho e avaliar as dependências que podem ser problemáticas na nova plataforma. O objetivo deste passo é obter uma visão completa do que deve ser mantido, atualizado, eliminado ou criado de raiz, facilitando uma migração mais limpa e organizada (Forbes, 2019).

2.9.3 Planeamento e Desenho da Migração

Uma vez que se tem uma compreensão clara da situação atual, o próximo passo é planear a migração. Isto envolve desenhar a nova estrutura do algoritmo/processo, levando em consideração a arquitetura, bibliotecas e ferramentas disponíveis da nova plataforma. Este desenho deve adaptar ou reescrever o algoritmo/processo para garantir que funcione corretamente na nova plataforma, podendo envolver alteração de bibliotecas, adaptação de funções e otimização do código para um melhor desempenho (Linkedin, s.d.).

2.9.4 Implementação inicial e Testes de Validação

Após definir a estratégia de migração, a implementação do algoritmo/processo pode começar. Durante esta fase, o código é transferido, adaptado e, se necessário, reescrito na nova plataforma. Todo o processo de implementação deve ser documentado cuidadosamente. À medida que se começa a migrar o processo, é essencial realizar testes de validação intensivos. Para isto, deve-se definir métricas de desempenho claras que permitam uma comparação objetiva dos resultados produzidos nas duas plataformas (Forbes, 2019) (Linkedin, s.d.). As métricas depende das especificidades do projeto em questão. No entanto, algumas métricas comuns que geralmente são consideradas incluem: tempos de execução, precisão dos resultados, capacidade de escalabilidade, custos, entre outros.

2.9.5 Implementação Final

Uma vez concluída a migração do algoritmo/processo para a nova plataforma, é necessário executar o processo em ambas as plataformas e validar os resultados. Se os resultados forem satisfatórios, a desativação da plataforma inicial pode ser iniciada (Linkedin, s.d.).

É importante salientar que a migração de algoritmos/processos entre plataformas não se limita apenas às etapas descritas de 2.9.1 a 2.9.5. Existem outros aspetos críticos como a gestão de risco, que envolve identificação e mitigação de potenciais riscos durante a migração, e a formação dos utilizadores, que assegura que estes estejam familiarizados e confortáveis com a nova plataforma. Estas considerações são fundamentais para uma migração bem sucedida, organizada e eficiente (Forbes, 2019).

No capítulo seguinte, irá ser abordado um caso prático de aplicação das boas práticas e metodologias previamente apresentadas na migração de um algoritmo do processo REACT no contexto da Sonae MC.

Capítulo 3

O Sistema REACT

O objetivo deste capítulo é fornecer uma explicação sobre o que é o sistema REACT, mostrando quais as suas funções e objetivos. Mais tarde o foco passará para o aspeto tecnológico do sistema, cobrindo a sua arquitetura, descrevendo de que modo é que o sistema está construído e fornecendo explicações sobre esta construção. Após esta introdução inicial ao tema, mostrar-se-á os estudos realizados para entender melhor o sistema. No final deste capítulo serão propostas melhorias para este sistema, com o objetivo de aumentar a sua eficiência e o seu desempenho em Databricks.

3.1 Descrição Teórica do Modelo

O sistema REACT surgiu em 2019. Nasceu de uma necessidade de suprir um problema da equipa de desenvolvimento comercial não alimentar da Sonae MC. Este problema está relacionado com produtos descontinuados ou produtos com uma quantidade elevada de *stock*. Estes tipos de produtos acabam por ser um problema, visto que os artigos descontinuados estão a ocupar espaços que poderiam estar a ser utilizados para artigos mais rentáveis e que trariam mais receitas para a empresa. Muitas das vezes, um artigo descontinuado é um artigo que acaba por gerar uma margem negativa, sendo muitas as vezes vendido só para ganhar espaço e para se conseguir reverter algum do dinheiro investido, em vez de ganhar lucro.

O sistema REACT foi desenvolvido para resolver esse mesmo problema, permitindo que a equipa do não alimentar gerisse de uma forma mais eficiente o *stock* de produtos. Este sistema visa uma gestão do ciclo de vida dos artigos baseada na monitorização ativa

e estratégias de baixa de preço. Esta gestão engloba 3 diferentes focos, que podem ir alternando ao longo do tempo:

1. *Foco em Performance*: Este foco tem como objetivo a monitorização contínua do desempenho do artigo, permitindo que os reabastecimentos de *stock* sejam realizados de forma mais eficiente. Isto é possível, pois o sistema leva em consideração um determinado período de vendas e faz uma previsão das vendas futuras. Conforme apresentado na Figura 3.1, o sistema envia alertas ao utilizador quando o número de vendas reais difere muito das vendas projetadas. Quando o número de vendas é inferior ao número projetado o sistema também faz sugestões de desconto.

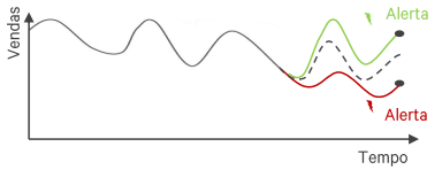


FIGURA 3.1: Foco em Performance

2. *Foco no Escoamento*: Este foco tem como objetivo a monitorização do artigo orientada para uma garantia de escoamento no período em questão. Isto é, pretende-se escoar o artigo, ficando com um *stock* limitado do artigo, num determinado período de vendas pré-definido pelo utilizador do sistema. Como se pode ver pela Figura 3.2, quando a quantidade de *stock* não está a baixar ao ritmo estimado, o sistema envia alertas para o utilizador. Se a taxa de escoamento for mais lenta do que o esperado, o sistema envia também uma sugestão de baixa de preço.

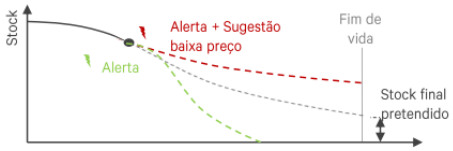


FIGURA 3.2: Foco no Escoamento

3. *Foco no Fim de Vida*: Este foco tem como objetivo a monitorização do fim de vida do artigo, garantindo o escoamento do *stock* e uma garantia de PVP adequado. O objetivo neste foco é ter *stock* zero do artigo. Como se pode ver pela Figura 3.4, o sistema envia um alerta ao utilizador quando o fim de vida do artigo é ultrapassado.

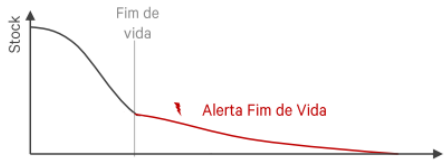


FIGURA 3.3: Foco no Fim de Vida

Neste sistema existem três tipos de artigos:

1. *Permanente*: Artigos que se prolongam em loja por um período longo de tempo, de forma permanente, e sem fim de vida previamente planeado. Um exemplo deste artigo é um copo simples que está o ano todo em loja.

2. *Recorrentes In-Out*: Artigos que permanecem em loja por um período limitado de tempo e que retornam à loja num período posterior. Um exemplo deste artigo são as árvores de Natal.

3. *Coleção*: Artigos que permanecem em loja por um período limitado de tempo, sem retorno previsto.

O sistema REACT atua em duas vertentes:

1. *Modelo de Ciclo de Vida do Artigo [CV]* - Com base nos dados históricos dos artigos, prevê-se o padrão de vendas futuras do artigo e as respetivas taxas de escoamento. Primeiramente usam-se modelos de *clusters* que agrupam e classificam artigos em grupos estatísticos com perfil de vendas semelhantes. O uso de *clusters* permite um melhor controlo e estabilidade sobre as previsões, em especial no caso de artigos novos. Mais tarde, são aplicados modelos de *machine learning*, tais como Modelos de Regressão, *Random Forest* e *Gradient Boosting Machine*.

2. *Price Elasticity of Demand* [PED] - Calcula a variação prevista na procura de um artigo resultante de uma baixa de preço, ou seja, avalia o impacto de diferentes descontos nas taxas de escoamento do *stock*. Para isto usam-se modelos de *clusters* que agregam os principais *price points* ao longo do histórico de preços do artigo, excluindo assim ligeiras variações observadas. Seguidamente, aplica-se modelos de regressão logística para estimar a elasticidade da procura-preço dos atributos.

3.2 Descrição Prática do Modelo

O sistema foi desenvolvido na plataforma Cloudera Machine Learning, em Python 3.6.

3.2.1 Arquitetura do Sistema

Para armazenar todos os dados necessários para as análises explicadas na Secção 3.1, o REACT utiliza o CML e o Hadoop. O processo inicia-se com a importação dos dados necessários para a execução a partir do Hadoop. Após esta importação, existe uma série de processamento e modelação de dados. Terminadas estas etapas, o CML envia os dados novamente para Hadoop. Por fim, a visualização dos dados e dos *dashboards* deste processo é realizado com recurso ao excel, onde este lê os dados que foram enviados para Hadoop, que são guardados em tabelas com recurso ao HDFS. O excel só por si não se liga diretamente ao Hadoop, o que existe atualmente é um *driver ODBC* *. A arquitetura é a seguinte:

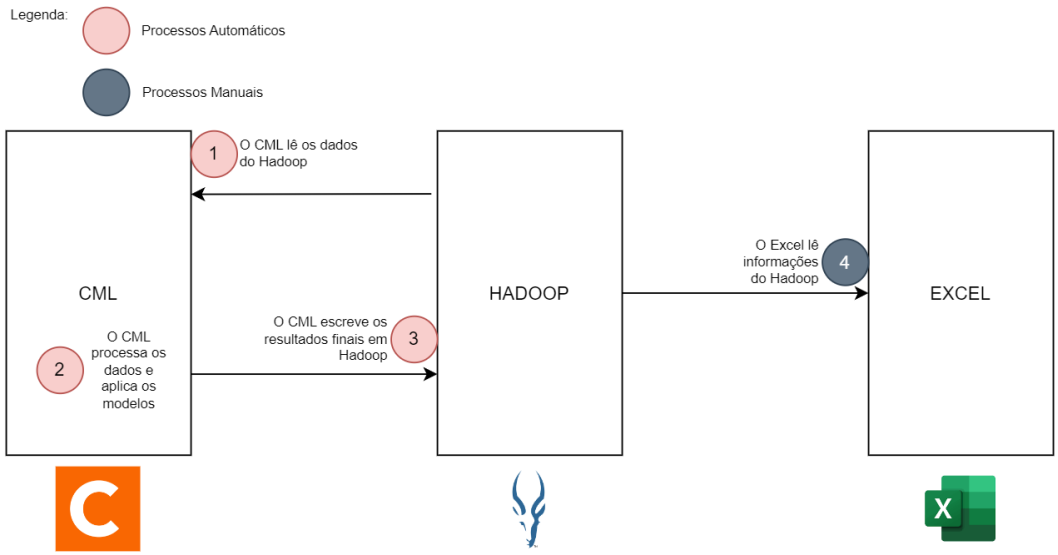


FIGURA 3.4: Arquitetura do Sistema REACT

*Um *driver ODBC (Open Database Connectivity)* é um componente de *software* que permite às aplicações, como por exemplo, o Excel, que interajam com vários sistemas de gestão de bases de dados, tal como o Hadoop/Impala.

3.2.2 Processos de Execução

Para executar o explicado na Secção 3.1, existem quatro processos principais, ou *pipelines*, de gestão e atualizações do sistema REACT:

1. *Processo de atualização semanal de indicadores e métricas*: Neste processo faz-se a atualização semanal da interface: inclui a atualização de indicadores, métricas fundamentais e dos modelos para a gestão e utilização do sistema REACT. É executada semanalmente às 14h de segunda-feira. Na Figura 3.5 mostra-se este processo. Cada retângulo corresponde a uma tarefa ou *job*, isto é, dentro de cada retângulo existe um conjunto de código que executa determinadas funções. Neste processo, quando se chega à tarefa [CV] Run Data Processing, o código é bifurcado e as restantes tarefas são corridas em paralelo. Quando um retângulo é executado, avança-se para o seguinte. O conteúdo entre parênteses retos refere-se a que tipo de trabalho cada tarefa tem: [HADOOP] refere que se está a importar ou a exportar dados para o Hadoop, [CV] quer dizer que os dados estão a ser utilizados para o modelo de Ciclo de Vida e o [PED] quer dizer que os dados estão a ser utilizados para o modelo *Price Elasticity of Demand*.

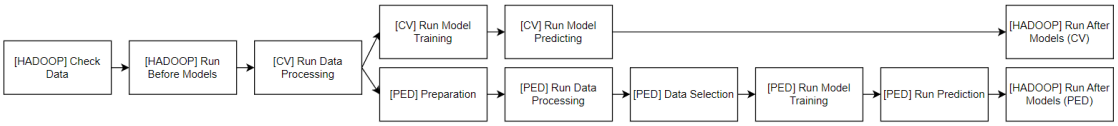


FIGURA 3.5: Processo de atualização semanal de indicadores e métricas

2. *Processo de treino do modelo de Price Elasticity of Demand*: Este processo surgiu devido ao elevado tempo de execução da *pipeline* anterior. Removeu-se o treino do modelo de *Price Elasticity of Demand*, executando-se à parte e inserindo esta informação no treino no processo anterior. Executa semanalmente, ao domingo, às 10h. No diagrama 3.6, mostra-se este processo.

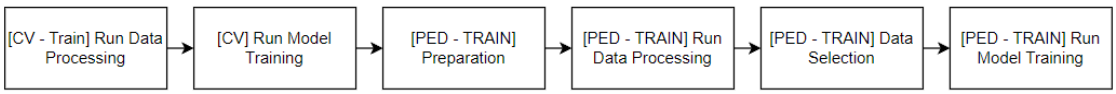


FIGURA 3.6: Processo de treino do modelo de Price Elasticity of Demand

3. *Processo de adição de novas categorias à estrutura REACT*: Para se adicionar novas categorias é necessário correr um processo de preparação das tabelas e modelos para as novas categorias. É executado de forma manual, sendo as categorias escolhidas à priori. No diagrama 3.7, mostra-se este processo.

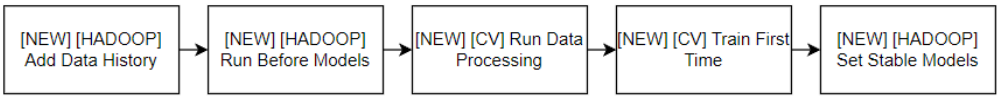


FIGURA 3.7: Processo de adição de novas categorias à estrutura REACT

4. *Processo de avaliação dos resultados de uma categoria*: Este processo é composto por uma única tarefa, como se mostra no diagrama 3.8. Foi desenhado para permitir a rápida avaliação de resultados de uma categoria numa determinada data histórica. Isto é, permite avaliar a qualidade do modelo, comparando as previsões do modelo com os dados reais. Este processo, tal como o anterior, também é corrido de forma manual. É importante salientar que a *pipeline* de avaliação dos resultados contém apenas uma tarefa, designada por 'Evaluate Models'. Esta tarefa engloba todo o código necessário para a comparação, incluí a preparação de dados e aplicação dos modelos de *machine learning*, idêntico ao empregado no processo de atualização semanal de indicadores e métricas. Inclui ainda código específico que possibilita a comparação entre os dados previstos e os dados reais.

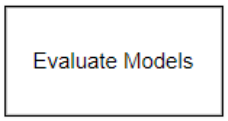


FIGURA 3.8: Processo de avaliação dos resultados de uma categoria

3.3 Exploração do Modelo

Antes de começar a migrar os algoritmos para Databricks, primeiro fez-se várias análises ao sistema de forma a entendê-lo melhor e a traçar possíveis otimizações e melhorias. Estas análises foram desde o tempo médio de execução das tarefas, às bibliotecas usadas, ao código e à organização dos ficheiros em CML. Vai-se, então, entrar em detalhe nas análises feitas ao sistema.

3.3.1 Análise de Tempos de Execução

De modo a entender o tempo total de execução e quais é que são as tarefas que demoram mais tempo, fez-se uma análise de tempos de execução na totalidade das tarefas de todas as *pipelines*.

A análise de tempos de execução é importante de se realizar antes de uma migração, pois é uma forma rápida de identificar gargalos no desempenho, isto é, partes do código que demoram muito a ser executados. Isto pode ser devido a vários motivos como, por exemplo, ineficiências no código. Estes gargalos também podem ajudar a orientar a migração, sugerindo mais esforços numa certa parte do código, ao invés de noutra.

Como dito anteriormente, o objetivo deste trabalho para além de migrar o processo para Databricks, também é otimizar o código que nele existe. Embora os tempos de execução não estejam correlacionados com a migração, mas sim com as características das máquinas, irá abordar-se esta análise para identificar potencias áreas de melhoria e otimização no código existente e, possivelmente, uma forma de comparação entre as duas plataformas.

Esta análise fez-se observando os tempos de início e de fim de cada tarefa dentro das *pipelines* durante algumas execuções e realizou-se um tempo médio de execução. Nas secções seguintes, de 3.3.1.1 a 3.3.1.4, encontram-se as Tabelas 3.1 a 3.4 com os tempos de execução para cada *pipeline* estudada anteriormente.

3.3.1.1 Processo de atualização semanal de indicadores e métricas

Desta análise, percebeu-se que as duas tarefas que demoram mais tempo a ser executadas são: [Hadoop] Run Before Models e [Hadoop] Run After Models (PED). Em ambas as tarefas existe a realização da execução de consultas SQL. Na primeira é onde se trazem os dados do Hadoop para o CML e na segunda é onde se enviam os dados dos modelos preditivos para Hadoop, para mais tarde serem mostrados em Excel.

Seguidamente, a terceira tarefa que demora mais tempo é [PED] Run Model Training, onde se executa o treino dos modelos *Price Elasticity of Demand*.

Desta análise retira-se que o estudo de otimização deve orientar-se mais para o lado de como se pode otimizar as consultas SQL e de como se pode otimizar o tempo de execução dos modelos de treino.

TABELA 3.1: Tempos de execução do processo de atualização semanal de indicadores e métricas

Processo de atualização semanal de indicadores e métricas	
Nome Tarefa	**Tempo Médio**
[HADOOP] Check Data	segundos - 10 min
[HADOOP] Run Before Models	6h00 - 7h00
[CV] Run Data Processing	30 min
[CV] Run Model Training	1h30
[CV] Run Model Predicting	1h00
[HADOOP] Run After Models (CV)	3 segundos
[PED] Preparation	segundos
[PED] Run Data Processing	1h00 - 2h00
[PED] Data Selection	4 min
[PED] Run Model Training	2h00 - 3h30
[PED] Run Predicting	1h00
[HADOOP] Run After Models (PED)	4h00 - 5h00

3.3.1.2 Processo de treino do modelo de Price Elasticity of Demand

TABELA 3.2: Tempos de execução do processo de treino do modelo de Price Elasticity of Demand

Processo de treino do modelo de Price Elasticity of Demand	
Nome Tarefa	**Tempo Médio**
[CV - TRAIN] Run Data Processing	30 a 40 min
[CV- TRAIN] Run Model Training	1h20 - 2h00
[PED - TRAIN] Preparation	segundos - 2 min
[PED - TRAIN] Run Data Processing	1h30 - 2h30
[PED - TRAIN] Data Selection	5 min
[PED - TRAIN] Run Model Training	2h00 - 4h00

Desta análise, percebeu-se que as tarefas que demoram mais tempo são as que processam os dados e que treinam os modelos de *machine learning*.

3.3.1.3 Processo de adição de novas categorias à estrutura REACT

Este processo é executado de forma manual quando existe a necessidade de se acrescentar novas categorias ao sistema, ou seja, o tempo de execução vai depender do número de categorias a importar e do tamanho histórico das mesmas. À semelhança dos processos anteriores, as tarefas que demoram mais tempo são as que vão buscar dados ao Hadoop.

TABELA 3.3: Tempos de execução do processo de adição de novas categorias à estrutura REACT

Processo de adição de novas categorias à estrutura REACT	
Nome Tarefa	**Tempo Médio**
[NEW] [HADOOP] Add History	3 dias
[NEW] [HADOOP] Run Before Models	1h00 - 8h00
[NEW] [CV] Run Data Processing	0h30 - 1h30
[NEW] [CV] Train First Time	0h10 - 1h30
[NEW] [HADOOP] Set Stable Models	segundos - 10m

3.3.1.4 Processo de avaliação dos resultados de uma categoria

TABELA 3.4: Tempos de execução do processo de avaliação dos resultados de uma categoria

Processo de avaliação dos resultados de uma categoria	
Nome Tarefa	**Tempo Médio**
Evaluate Models	1h00

À semelhança do processo anterior, este também é executado de forma manual. Avalia-se resultados de categorias; o tempo de execução também dependerá do número de categorias existentes.

3.3.2 Análise de Bibliotecas

Seguidamente, analisou-se as versões das bibliotecas usadas neste projeto e se estas eram ou não nativas do Python. Este processo é fundamental pois, as bibliotecas, quer sejam nativas ou não, podem ser atualizadas, isto é, de uma versão para a outra pode retirar-se uma determinada função utilizada ou acrescentar o número de parâmetros usados noutra. Relativamente às bibliotecas não nativas, estas ainda correm o risco de não estar disponíveis em todas as versões da linguagem.

Em conclusão, o estudo das versões das bibliotecas e se estas são ou não nativas é uma etapa crítica de um processo de migração, permitindo garantir a compatibilidade do código e identificar possíveis problemas e melhorias de desempenho, facilitando a migração e evitando trabalho desnecessário.

Na Tabela 3.5 mostra-se as versões das bibliotecas utilizadas neste projeto e se são ou não nativas:

TABELA 3.5: Versões das bibliotecas utilizadas e se são ou não nativas do Python

Biblioteca	Versão	É nativa?
sys	3.3	Sim
subprocess	3.5	Sim
time	3.3	Sim
warnings	3.2	Sim
pandas	1.0.3	Não
loguru	0.4.1	Não
joblib	0.14.1	Não
importlib	3.3.0	Sim
h2o	3.28.0.3	Não
datetime	3.6	Sim
ibis.impala	3.0.1	Não
impala.dbapi	1.3.0	Não
sklearn	0.22.1	Não
pyarrow.parquet	0.16.0	Não
statsmodels	0.11.1	Não
seaborn	0.10.0	Sim
matplotlib	3.1.3	Não
tqdm	4.43.0	Não

3.3.3 Análise do Código

O seguinte passo foi estudar o código na sua totalidade, o que é que cada tarefa realiza e que dados é que são utilizados. Este passo é muito importante, pois é nele que se vai entender o processo. Uma migração sem noção de todo o código poderá trazer problemas mais à frente. Com o estudo realizado de antemão, compreende-se melhor como é que o código e a arquitetura funcionam. Visto que o processo de migrar um código pode ser complexo e à medida que se migra podem surgir problemas, esta etapa é fundamental, pois pode-se aperceber de algum problema antes da migração e abordá-lo antes de começar a mesma. Também é uma ótima maneira de verificar se existem redundâncias ou partes do código que podem ser otimizadas, levando a uma melhoria de desempenho e eficiência.

Após o estudo das *pipelines* do projeto e da duração média de cada tarefa, ficou-se com uma ideia de onde é que os 'gargalos' do problema se encontram. Um gargalo pode ocorrer em diferentes partes, podendo ser identificado ao observar onde ocorre uma maior lentidão no sistema. Por exemplo, se um sistema que processa um grande volume de dados apresenta uma lentidão na fase de leitura, pode-se concluir que há um gargalo nessa operação.

Nas subsecções seguintes 3.3.3.1, 3.3.3.2, 3.3.3.3 e 3.3.3.4 irá-se mostrar diagramas conceptuais dos que se encontram na Secção 3.2.2, isto é, uma explicação do que cada tarefa ou conjunto de tarefas executa. Também serão acrescentados alguns comentários que se ache pertinente à interpretação da execução das mesmas.

3.3.3.1 Processo de atualização semanal de indicadores e métricas

Nesta *pipeline* existe uma validação da existência de datas específicas da informação, ou seja, o sistema liga-se às bases de dados em Hadoop e procura se as datas pretendidas já se encontram disponíveis nas tabelas. Caso estas já estejam disponíveis, estas são migradas do Hadoop para o CML, caso contrário, a execução é abortada. Seguidamente, existe uma bifurcação no código, que pode ser observado no diagrama conceptual 3.9, no ramo de cima, onde se vai treinar e aplicar os modelos de ciclo de vida. No ramo de baixo, vai-se treinar e aplicar o modelo de elasticidade de preço. Quando estes dois modelos se encontram terminados, atualiza-se as tabelas de pós-processamento, enviando dados para o Hadoop novamente. Conceptualmente, pode-se representar este parágrafo com o diagrama 3.9:

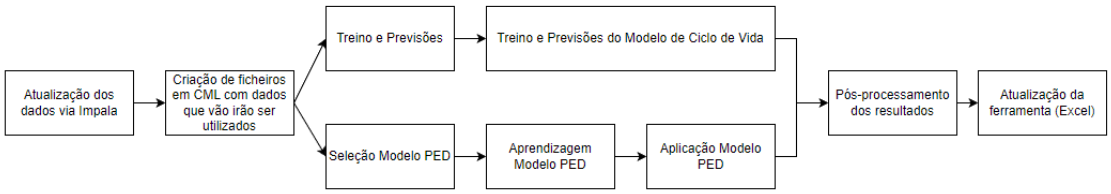


FIGURA 3.9: Diagrama Conceptual do diagrama 3.5

3.3.3.2 Processo de treino do modelo de Price Elasticity of Demand

Nesta *pipeline*, existe o processamento das variáveis que foram puxadas do Hadoop para o CML através da *pipeline* anterior, sendo treinado o modelo de elasticidade do preço. O diagrama 3.10 faz um resumo dos passos:

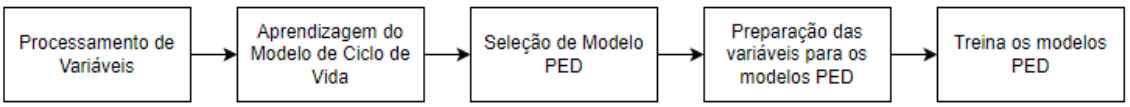


FIGURA 3.10: Diagrama Conceptual do diagrama 3.6

3.3.3.3 Processo de adição de novas categorias à estrutura REACT

Nesta *pipeline*, existe a adição de novas categorias que não se encontram presentes em CML. Este liga-se ao Hadoop para verificar quais é que são as categorias que ainda não foram adicionadas, sendo então a informação relativa a estas novas categorias puxada para CML. Seguidamente, as tabelas são atualizadas e o sistema faz um processamento de informação para cálculo e afinação dos modelos, sendo construído o primeiro modelo de ciclo de vida para as novas categorias. Após este treino, as novas categorias estão prontas para integrar a *pipeline* de atualização semanal de indicadores e métricas. No diagrama 3.11, resume-se estas atividades:

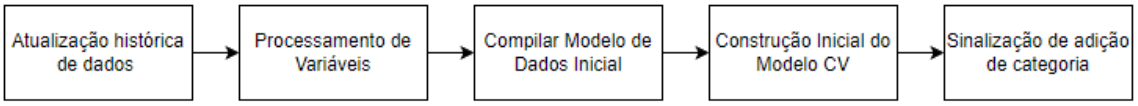


FIGURA 3.11: Diagrama Conceptual do diagrama 3.7

3.3.3.4 Processo de avaliação dos resultados de uma categoria

A última *pipeline* serve para avaliar a qualidade dos modelos de uma determinada categoria, ou seja, faz-se uma comparação entre as previsões realizadas com as vendas reais. Concetualmente, a tarefa é representada pelo diagrama 3.12:

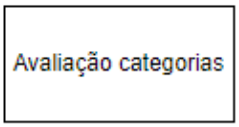


FIGURA 3.12: Diagrama Conceptual do diagrama 3.8

3.3.4 Realização de Diagramas

Após estudar o código e de já se ter uma compreensão melhor do processo e das suas tarefas, passou-se para a discussão de melhorias na migração. Como nesta migração se vai utilizar a plataforma Databricks, que é uma plataforma que aproveita o paralelismo do Apache Spark, uma das primeiras melhorias sugeridas envolveu a utilização de mais paralelismo.

Embora no CML o único paralelismo existente seja na *pipeline* de atualização semanal de indicadores e métricas, este é um paralelismo apenas relativo a tarefas. A ideia das melhorias seria introduzir mais paralelismo onde atualmente não existe, isto é, nas consultas de SQL e no código. Paralelizar a execução de várias categorias, ao invés do que acontece atualmente: sequencialmente.

De modo a determinar que partes do código é que podem ser paralelizadas, prosseguiu-se para a realização de diagramas, visto que é uma maneira visual de entender o fluxo de código e entender que partes é que podem ser paralelizadas e que partes é que não podem ser.

Para as tarefas foram criados vários diagramas de funções e de ficheiros *input's* e *output's*. Com estes diagramas realizados, a tarefa de decidir o que é possível paralelizar torna-se muito mais fácil. Devido à elevada complexidade dos diagramas, estes irão ser divididos em diagramas mais pequenos e de modo a entendê-los melhor, segue uma breve explicação sobre como os interpretar: Os paralelogramos significam que são ficheiros de *input's* e de *output's*. Os que estão pintados de amarelo são ficheiros de *input ´s* que vêm de outra Figura, enquanto que os paralelogramos vermelhos são ficheiros de *output ´s* que vão ser utilizados como *input ´s* noutra função. Os retângulos em 3D são o nome das tarefas dos diagramas, como por exemplo: [CV] Run Model Predicting. Os retângulos que estão pintados a branco são as funções que estão dentro dessa tarefa. A numeração indica qual é a ordem das funções dentro dessa tarefa.

Para as *queries*, fez-se um trabalho com uma lógica similar: as tabelas que vão ser criadas ao longo do processo vão estar identificadas com um número entre parêntesis reto, que indica a ordem de execução, como por exemplo: react_product [1]. O sistema de cores é o mesmo, *queries* a vermelho quer dizer que vão ser utilizadas numa outra mais à frente e uma *query* amarelo quer dizer que vem de outro diagrama anterior. Relativamente às tabelas, existem 3 formas de como as *queries* são alimentadas. Uma dessas formas é utilizar tabelas fornecidas pela Sonae, as quais estarão a verde. Uma outra forma é de tabelas de *input* manual, como por exemplo: ficheiros CSV, estarão a azul. Por último, a cinzento estarão as que, através de tabelas criadas especificamente para o processo, serão designadas como tabelas REACT.

Nos anexos A.1, A.2, A.3 e A.4 mostra-se os diagramas que se realizou para entender melhor o fluxo do código dos diagramas que se encontram na Secção 3.2.2. Estes diagramas de fluxo são fundamentais para entender que partes do código é que podem ser

paralelizadas.

Os diagramas detalhados do plano de execução das *queries* de pré-processamento dos dados encontram-se no anexo A.5, juntamente com uma legenda de como interpretar as diferentes cores.

Após uma análise detalhada dos diagramas das tarefas é possível denotar alguns detalhes sobre o processo atual em CML. Por exemplo, no diagrama 3.13, torna-se claro que as funções *predict_clustering* e *train_classification* são candidatas ideais para paralelização. Isto deve-se ao facto de operarem independentemente uma da outra, ou seja, o *input* de uma não resulta do *output* da outra. Nota-se pela localização de onde os dados são guardados (representado nos paralelogramos a vermelho) que os modelos processam treino e previsões de categorias de forma sequencial. No entanto, é fundamental sublinhar que os dados de uma categoria não influenciam, em momento algum, o treino e previsão de outra. A independência entre as categorias reforça a viabilidade de um processamento em massa. Além disso, observando os diagramas, é possível reparar na quantidade de ficheiros existentes na plataforma CML, o que sublinha a importância de otimizar este processo.

No contexto deste processo, há operações que necessitam de se manter numa abordagem sequencial para assegurar a integridade e precisão dos resultados. Voltando novamente ao diagrama 3.13, é evidente que as funções *train_clustering* e *predict_clustering*, responsáveis pelo treino e a previsão de *clustering*, exigem essa sequencialidade. Mesmo com a oportunidade de paralelizar certas funções ou processar dados em massa, é necessário reconhecer que determinados procedimentos necessitam de ser sequenciais.

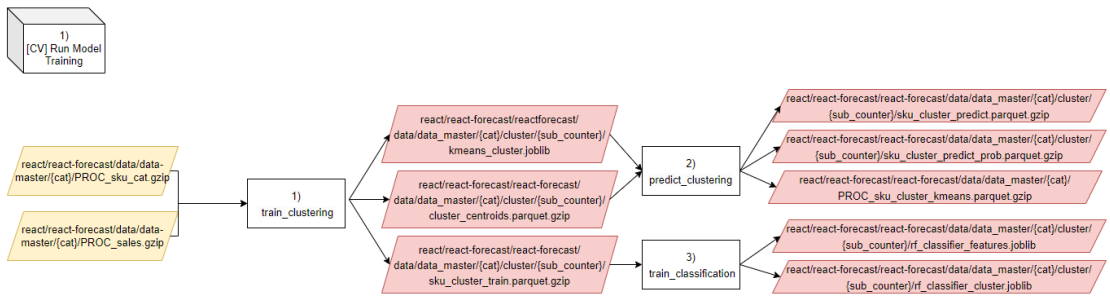


FIGURA 3.13: Diagrama do fluxo de código do ramo de cima - Parte I da *pipeline* 3.5

De forma semelhante, analisando o diagrama 3.14, consegue-se perceber quais *queries* é que podem ser executadas em paralelo. Por exemplo, as tabelas *react product* (tabela com os detalhes dos artigos a considerar para o processo), *react location* (tabela com os detalhes

das lojas a considerar para o processo), e *react category location* (tabela com informação da insignia a considerar para cada categoria) podem todas ser criadas em paralelo.

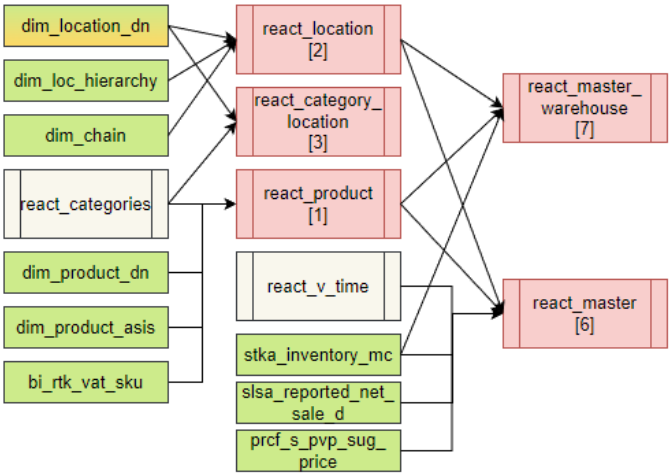


FIGURA 3.14: Diagrama do fluxo das *queries* 3.8 - Parte I

Após o estudo que acabou de se mostrar, seguiu-se para o levantamento de melhorias que se acha possível realizar na migração do processo de CML para Databricks.

3.3.5 Melhorias Propostas

Após o estudo descrito nas secções anteriores, já se possui um melhor entendimento do processo e dos seus fluxos. À medida que foi realizada esta análise foi-se anotando pontos de possível otimização e melhoria, tanto no código como na estrutura atual do sistema em CML. A migração de CML para Databricks visa melhorar os pontos referidos anteriormente, bem como melhorar a eficiência e a escalabilidade do projeto. As melhorias sugeridas para esta migração foram:

1. Ao longo do estudo realizado, apercebeu-se que a *pipeline* de treino do modelo de *Price Elasticity of Demand* é redundante. Como dito anteriormente, a ideia deste processo é aliviar o tempo de execução da *pipeline* de segunda-feira, sendo realizado ao domingo. Contudo, este treino ainda acontece todas as segundas-feiras, duplicando o tempo investido sem utilizar os dados do treino de domingo. Por gastar mais tempo e recursos sem necessidade para tal, esta pipeline não será migrada para a nova plataforma;

2. Ao longo deste estudo também se observou alguma redundância na parte de treino do modelo de Ciclo de Vida. Atualmente, este modelo repete o treino, sendo um

desperdício de tempo. Por este motivo, vai-se remover estas redundâncias, de forma a manter o código mais limpo, mais fácil de ler e mais fácil de se manter;

3. Como visto anteriormente, o processamento de consultas corresponde à tarefa que demorava mais tempo. Uma das razões para isso acontecer é a sequencialidade da sua execução. Pretende-se então nesta migração a realização do processamento de consultas (*queries*) em *bulk*, permitindo uma redução do tempo de execução desta etapa;

4. Como já comentado, o treino dos modelos é realizado de forma sequencial, categoria a categoria, sendo que no futuro se pretende uma maior paralelização deste treino;

5. Atualmente algumas tabelas estão a ser particionadas ao dia e pela categoria. De modo a ter consultas mais eficientes, pretende-se particionar as tabelas apenas pelo dia;

6. Ao longo do código, a biblioteca Pandas do Python é utilizada com muita frequência. Na migração pretende-se reduzir o uso desta biblioteca e utilizar PySpark, aproveitando-se assim do poder de processamento distribuído e paralelo oferecido pelo Apache Spark em Databricks;

7. Utilizar Spark para ler e escrever os dados, permitindo assim o processamento paralelo e distribuído. Desta forma pretende-se melhorar o desempenho e a escalabilidade do projeto;

8. Simplificar a estrutura atual, reduzindo a quantidade de ficheiros existentes na plataforma;

9. Orquestração dos *pipelines* em Azure Data Factory.

Capítulo 4

Implementação do REACT em Databricks

Este capítulo tem como objetivo mostrar o processo de migração do sistema REACT da plataforma Cloudera Machine Learning para Databricks. Ao longo deste capítulo vai-se abordar o planeamento e a execução da migração, partilhando alguns detalhes técnicos e desafios encontrados pelo caminho. Vai-se começar pela *pipeline* de atualização semanal dos indicadores e métrica, visto que é o principal processo deste sistema. Este processo, embora complexo, representa uma oportunidade de melhoria em termos de eficiência operacional, escalabilidade e potencialidade de inovação.

4.1 Introdução à migração

Para se começar a migrar o sistema foi necessário a criação de um ambiente e de um cluster em Databricks, contudo esta parte não se focará nos detalhes e será assumido que existe um ambiente produtivo e um cluster funcional.

Após o estudo realizado no capítulo anterior, percebeu-se que a *pipeline* mais importante para o processo era a da atualização semanal de indicadores e métricas (Figura 3.5), fornecendo semanalmente dados para a equipa do não alimentar da Sonae, possibilitando a tomada de decisões em relação a preços e descontos.

4.2 Organização do código

Antes de começar a falar da migração, vai-se falar um pouco sobre a organização dos ficheiros e do código em Cloudera e sobre qual será o plano da organização futura.

4.2.1 Organização em Cloudera

Em Cloudera existem quatro pastas principais: uma de *output's*, que contém os modelos treinados e as previsões; e três pastas, denominadas como *react forecast*, *react hadoop* e *react ped* referentes aos três tipos diferentes de utilização de dados, como explicado em 3.2.2. Dentro de cada uma destas três pastas estão ficheiros de conFiguração, ficheiros de dados de input manual, planos de execução de código Python, *queries* para processar dados, entre outra informação.

Por exemplo, dentro da pasta *react hadoop*, existe outra pasta chamada *queries* e por sua vez, dentro desta existem mais do que 30 ficheiros de SQL referentes às consultas de dados de pré e pós processamento.

O processo atual em CML possui várias pastas dentro de pastas, aumentando assim a complexidade da navegação e da localização de informações específicas. O objetivo da migração, para além de ser implementar o processo em Databricks, também é de certa forma simplificar a estrutura atual.

4.2.2 Organização em Databricks

Em Databricks vai-se implementar o sistema de organização de uma forma um pouco diferente. Em vez das quatro pastas como em CML, existirão três pastas: uma chamada de *scripts*, que contém ficheiros auxiliares, como por exemplo o *notebook* que cria o formato das tabelas; outra denominada de *modules*, que por sua vez tem mais três pastas dentro (*react ped*, *react forecast* e *react hadoop*), que contém os *notebook*s que vão ser executados na *pipeline*, pelo Data Factory; e por fim *config*, onde estão localizados todos os ficheiros de conFiguração para o processo.

4.3 Processo de atualização semanal de indicadores e métricas

Visto de uma forma genérica o método de organização dos *notebook* e de ficheiros, vai-se avançar para a migração de código da primeira *pipeline* do sistema. Primeiramente, mostrar-se-ão melhorias acrescentadas ao processo, depois avançando-se para as tarefas.

4.3.1 Melhorias em Databricks

Devido à complexidade e extensão do projeto, a identificação de erros de execução, ou seja, quando o processo falha por algum motivo, pode apresentar ser uma tarefa desafiadora. Na plataforma CML havia a prática de utilização do recurso de *logging* através da *loguru*. Esta biblioteca não é nativa ao Python e existe neste processo a consciência da importância de minimizar a dependência deste tipo de bilbiotecas, uma vez que podem introduzir potenciais vulnerabilidades, reduzir a eficiência do código e complicar a manutenção do mesmo. Com estas considerações em mente, e com o intuito de assegurar a robustez e simplicidade da solução, optou-se, na plataforma Databricks, pelo desenvolvimento de uma classe de *logging* cujo *design* tem como objetivo a clareza e facilidade de compreensão. O padrão adotado para estes registos é estruturado da seguinte forma: horário de ocorrência - categoria do *logging* (como, por exemplo, ser *warning, info, debug,* ...) - o nome da função - e a mensagem específica associada ao *logging*. Na Figura 4.1, encontra-se um exemplo da estrutura do *logging*. Como se pode ver, o *logging* foi realizado às 18h16m35s do dia 21-08-2023, surgiu da função *run table query* (que é uma função que escreve a *query* na sua tabela) e a *query* afetada foi a react models input.

```
[2023-08-21 18:16:35] [INFO] [Function: run_table_query] - [We are quering react_models_input]
```

FIGURA 4.1: Exemplo de uma estrutura de *logging*

Na tentativa de modernizar e otimizar os processos envolvidos na migração, foi desenvolvida uma nova funcionalidade que envolve um envio de uma notificação às pessoas responsáveis do estado da *pipeline*. Se esta falhar, o sistema envia automaticamente um e-mail detalhando o erro em questão e indicando a sua origem. Por outro lado, se a execução da *pipeline* for bem sucedida, será enviado um e-mail a indicar isso mesmo. Esta nova funcionalidade visa não só a eficiência operacional do projeto, mas também a garantia de uma resposta rápida e informada a possíveis desafios.

4.3.2 [Hadoop] Check Data

Nesta tarefa específica, não se realizou grandes alterações. Existe a criação de variáveis, com recurso a Python, dos dias e filtros necessários, como por exemplo, o dia de hoje, o dia do último domingo, o filtro do mês (que seria como: 202307). Depois desta definição, é usado SQL para verificar se as datas estão ou não nas tabelas. Se estas não estiverem, o *notebook* é abortado (e consequentemente a *pipeline*).

A única grande alteração que se fez nesta tarefa foi acrescentar um widget (ou um parâmetro) no *notebook* em Databricks, chamado de *Run ByDF*, com um valor *default* igual a *False*, como se pode ver na Figura 4.2

FIGURA 4.2: Widget default do *notebook* Check Data

Ou seja, em Data Factory este *notebook* é definido com o parâmetro do widget para *True*, como se vê na Figura 4.3. Isto quer dizer que sempre que a *pipeline* é executada, este *notebook* é executado e se as datas estiverem nas tabelas, avança-se para a próxima etapa, se não, a *pipeline* é abortada. Quando se executa este *notebook* só por si, ele tem o widget como *False*, ou seja, apenas o *notebook* é executado.

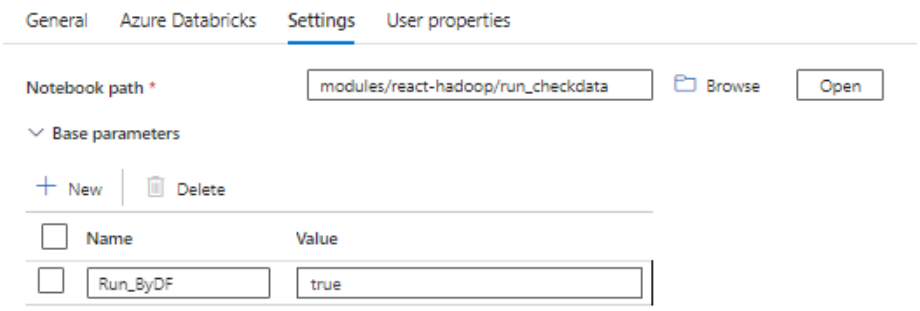


FIGURA 4.3: Widget default do *notebook* Check Data

4.3.3 [Hadoop] Run Before Models

Seguidamente avançou-se para a migração da tarefa da criação das tabelas do pré-processamento. Só nesta etapa são executadas 21 *queries*.

Em CML a execução desta tarefa é orquestrada através de um *notebook*. Esta execução de tarefas é realizada de forma sequencial, seguindo uma abordagem ordenada. Dentro deste *notebook* existe a função principal *go()*, que é responsável por coordenar a execução de diversas classes de funções, tais como:

1. ***Transaction***: é responsável pelo encapsulamento da execução; se esta falhar, a classe tentará novamente até atingir um número máximo de tentativas pré-definido anteriormente;

2. ***Drop***: é responsável pela remoção da tabela da execução anterior;

3. ***Query***: é responsável pela consulta dos dados, recebe o caminho onde a *query* se encontra e vai buscá-la; mais tarde o comando *.execute()* (Figura 4.4) executa esta consulta utilizando um cliente impala;

4. ***ComputeStats***: é responsável pelo cálculo de estatísticas das tabelas.

Cada uma destas classes representa uma etapa específica do processo e executa as *queries* de acordo com a ordem determinada, como se pode ver na Figura 4.4.

```
def go():

    cats = hd.get_react_categories(data=False, models = False)

    if len(cats) > 0:

        logger.info('Found {} new category/ies'.format(len(cats)))

        Transaction(
            [
                Drop('react_product'),
                Query(filepath=f'{PATH_QUERIES_INSERTS}/01_react_product.sql'
                    #   ,variables=OPTIONS
                    ),
                ComputeStats('react_product')
            ]
        ).execute()

        Transaction(
            [
                Drop('react_location'),
                Query(f'{PATH_QUERIES_INSERTS}/02_react_location.sql'),
                ComputeStats('react_location')
            ]
        ).execute()
```

FIGURA 4.4: Extrato do *notebook* que realiza o pós-processamento

Para migrar para Databricks é necessário criar as tabelas nesta plataforma e depois orquestrar o processamento das *queries*, tendo sempre como objetivo um output igual em Databricks e em CML. As tabelas criadas serão deltas (2.6.3), devido às suas inúmeras vantagens, tais como a compactação dos dados e a sua escalabilidade. A metodologia de comparação entre as tabelas é descrita na Secção 5.1.

Foi criado um *notebook* chamado *create table* que contém os *scripts* de criação das tabelas. Estas novas tabelas terão, inicialmente, um prefixo *db* para se facilitar a comparação com as tabelas em CML. É importante relembrar que esta migração está a ocorrer concomitantemente com a execução do processo REACT em CML. Assim, enquanto a migração está em curso, a plataforma Cloudera permance operacional. A escolha do prefixo *db*, que simboliza Databricks, visa, precisamente a comparação eficaz entre os resultados nas duas plataformas. Mais tarde, quando o processo estiver estabilizado este prefixo será deixado. Juntamente com a instrução *CREATE TABLE*, também serão utilizadas propriedades do Delta Lake para otimização da escrita das tabelas. Na Figura 4.5, pode-se ver o exemplo de um *script* de criação de uma tabela em databricks. As propriedades são:

1. ***optimizeWrite***: habilita a otimização automática ao escrever dados em tabelas delta, ajusta automaticamente o *layout* dos ficheiros para melhorar o desempenho de leitura e reduzir o tamanho do armazenamento;

2. ***autoCompact***: ativa a compactação automática dos ficheiros em tabelas delta, monitorizando e reorganizando-os para melhorar o desempenho de leitura e escrita.

```
1  CREATE TABLE db_tabela_CML (
2    coluna type
3  ) using delta TBLPROPERTIES (
4    delta.autoOptimize.optimizeWrite = true,
5    delta.autoOptimize.autoCompact = true
6  ) LOCATION 'path';
```

FIGURA 4.5: Exemplo da criação de uma tabela em Databricks

Seguidamente, passou-se para a migração das *queries* propriamente ditas. Em CML, estas encontravam-se numa pasta denominada por *queries* e dentro desta pasta encontravam-se os ficheiros com as consultas individuais. O que existe em Databricks é a existência um *notebook* denominado por *script_queries*, que conterá todas as *scripts* de pré e pós-processamento, juntamente com os seus ficheiros de conFiguração, como se pode ver na Figura 4.6.

Cada *query* terá associadas duas células do *notebook*, uma com o nome da tabela seguido de um '_cfg' (exemplo: tabela_cfg), contendo o ficheiro de conFiguração da *query*. Este ficheiro contém informações sobre como é que a escrita da *query* na tabela será realizada, incluindo detalhes sobre particionamento, caso aplicável, e a variável de partição, entre outras informações relevantes.

A segunda célula, vai ter o nome da tabela seguida de um '_query' (exemplo: tabela_query) que contém a *query*. Na Figura 4.6, encontra-se um exemplo de como estas duas células se parecem em Databricks.

Ao migrar o SQL das *queries* para Databricks (que utiliza o Spark SQL), definiu-se que se queria manter o mesmo output tanto em Databricks como em CML. Além disso, foi determinado que deveria ser realizado um estudo das *queries* para identificar redundâncias de código e pontos de melhoria. O objetivo principal é otimizar o desempenho e a eficiência das *queries* durante a execução. De seguida, serão enumerados alguns problemas/alterações que se realizaram nas *queries*:

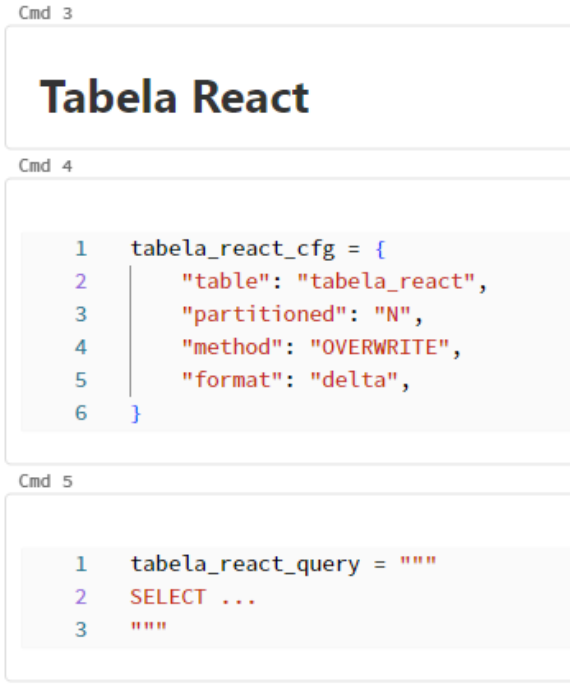


FIGURA 4.6: Exemplo de conFiguração de uma *query* em Databricks

1. Foi necessário adaptar algumas funções que funcionavam em Impala, mas que não eram compatíveis em Spark SQL. Alguns exemplos destas funções são: *FROM_TIMESTAMP()*, que foi substituída por *DATE_FORMAT()*, que são usadas para devolver a data no formato *yyyyMMdd*. E a função *APPX_MEDIAN()*, que é uma função de agregação que retoma um valor que é aproximadamente a mediana, foi substituída por *PERCENTILE(,0.5)*, que retoma o valor do percentil 50, que representa a mediana;

2. Removeram-se algumas redundâncias, tendo por exemplo um caso em que existia a instrução *WHERE*, filtrando dados, com datas superiores a 2016, contudo na tabela em questão não existiam datas inferiores a 2017;

3. Foi feito um estudo inicial do fluxo das *queries* e observou-se que a tabela *react sku promo* (mostrada na Figura A.27) era apenas utilizada na tabela *react models*. Por esse motivo, foi realizado um estudo comparativo para analisar o desempenho entre criar a tabela *react sku promo* separadamente versus incorporar a lógica da *query* que alimenta essa tabela na lógica da *react models*. Concluiu-se que a segunda opção apresentou um desempenho mais rápido e eficiente;

4. Existia uma tabela que estava a ser criada recorrendo a três *queries* distintas. Após realizar um estudo para analisar as dependências entre estas três *queries*, chegou-se à conclusão de que seria viável unir numa só *query*;

5. Ao longo desta etapa também havia tabelas de *input* manual, ou seja, ficheiros CSV com parametrizações importantes para o projeto. Em Databricks seguiu-se a mesma lógica, inserindo-se estes ficheiros CSV na plataforma.

Finalmente, foi criado um *notebook* genérico denominado de *run queries*, que recebe dois parâmetros dinâmicos: "Run_ByDF", com a mesma lógica da tarefa anterior (ver Figura 4.2) e, "Tabela", que representa a tabela a ser criada, como se pode ver na Figura 4.7. Este *notebook* contém uma função chamada de *run_table_query*, responsável por buscar o dicionário de conFiguração e a *query* correspondente à tabela que se introduziu no *widget*, executando a leitura da *query* e a escrita da mesma numa tabela.

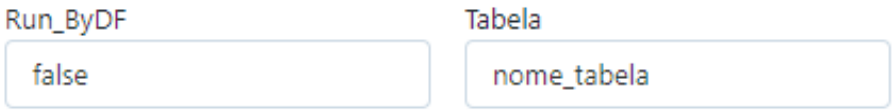


FIGURA 4.7: *Widget's* da tarefa *run queries*

Enquanto a migração se encontra ativa, também existe um *notebook* chamado *teste queries*, onde são realizados os estudos comparativos entre as duas tabelas. Estes estudos têm como objetivo verificar se ambas as tabelas produzem o mesmo número de linhas e se o conteúdo entre as duas tabelas é idêntico.

Após um estudo das dependências das *queries*, orquestrou-se a execução das mesmas em Azure Data Factory (2.8.1), como se pode ver nas Figuras 4.8 e 4.9:

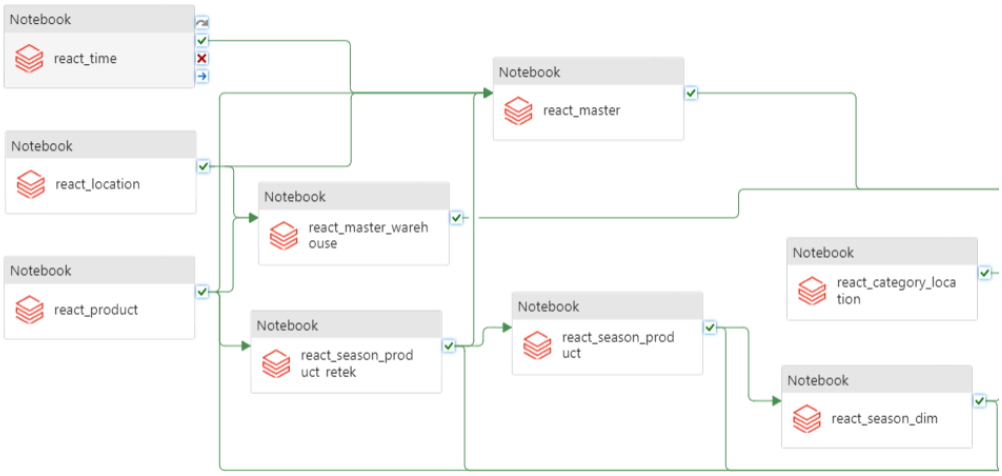


FIGURA 4.8: Orquestração do Run Before Models em ADF - I

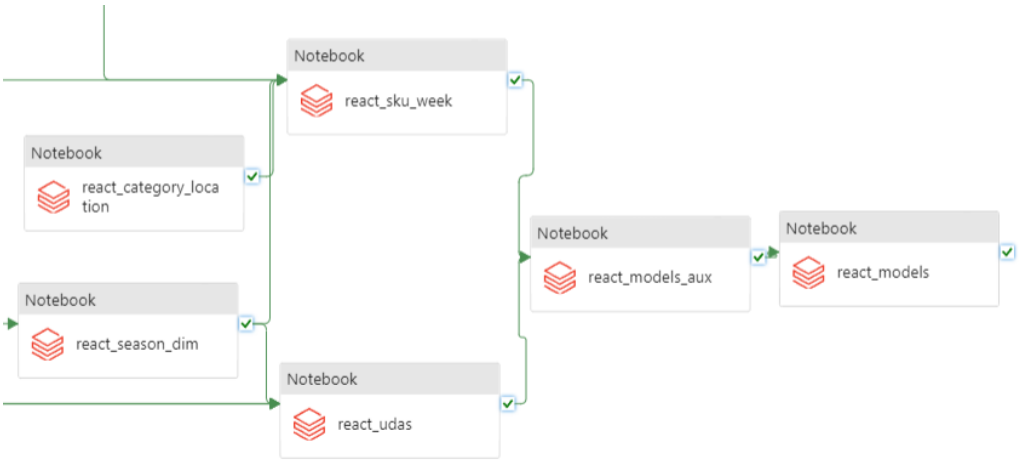


FIGURA 4.9: Orquestração do Run Before Models em ADF - II

4.3.4 [CV] Run Data Processing

Esta tarefa é executada a seguir à criação das tabelas de pré-processamento e existe a realização de três funções, que na sua essência fazem tratamentos, filtragem e limpeza de dados seguido de *feature engineering**. Nos parágrafos seguintes explica-se tanto as três funções previamente nomeadas como o trabalho executado durante a migração.

Na primeira função, denominada de *run_data_loading*, observa-se um processo iterativo de categoria a categoria do carregamento e transformações de dados, utilizando a biblioteca Pandas. Primeiramente importa-se um ficheiro *.txt* que equivale às linhas da

**Feature Engineering ou Engenharia de Características, que se refere ao processo de criar ou modificar variáveis para melhorar a adequação dos dados para modelagem, análise ou outras operações de dados.*

tabela *react_models*, consolidando-o num *DataFrame* para a categoria especificada. Seguidamente, existe a remoção de linhas duplicadas, remoção de produtos que não têm associado nenhuma *season*, inferência de tipo de dados pelo Pandas e tratamento de datas. Também existem duas condições específicas que, se caso sejam satisfeitas, ativam procedimentos de filtragem de dados, como por exemplo, dividir o *DataFrame* numa data específica. No final, existe a exportação do novo *DataFrame* para o formato Parquet.

No ambiente Databricks, optou-se por não migrar esta função na sua forma original. Após uma análise cuidadosa, observou-se que todas as ações realizadas pela função, desde a remoção de linhas até ao tratamento das colunas de datas, podiam ser eficientemente incorporadas na própria *query* da criação da tabela *react_models*. Considerando que esta tabela não será utilizada em qualquer outra etapa para além da atual, a decisão foi ajustar e centralizar a lógica diretamente na *query*, otimizando assim o processo. No pedaço de código seguinte 4.1, mostra-se um pouco de como é que se incorporou a lógica da função *run_data_loading* na *query*.

```
SELECT * FROM react_models_cte WHERE sku NOT IN (
    SELECT sku
    FROM (
        SELECT sku, coalesce(max(season_phase_id),0) as season_phase_id
        FROM react_models_cte
        GROUP BY sku
        ) f
    WHERE season_phase_id = 0 )
    AND
    CASE
        WHEN {predict_only} = 'Y' then uda3 = 'No'
        WHEN {predict_only} != 'Y' THEN True
        ELSE True
    END
AND time_key < {date}
```

CODE LISTING 4.1: Exemplo de otimização da função para uma query

Seguidamente, temos a segunda função denominada de *run_data_processing* responsável por diversas etapas de processamento e preparação de dados, assim como pela engenharia de características. Dentre as operações realizadas, alguns exemplos são: a determinação do período de sortimento ativo * baseado em vendas e critérios sazonais, resultando

*Sortimento ativo no mundo do retalho refere-se à seleção atual e disponível de produtos que uma loja oferece aos seus clientes num determinado momento

na criação de uma coluna binária que indica se um produto estava ativo ou não num determinado período; o cálculo do número de vezes que um sku entrou numa estação; a criação de uma coluna com a data mínima de vendas e a extração de mês e ano; a criação de métricas estatísticas, como a média e o desvio-padrão, entre outras transformações. A função conclui com a exportação de dois *DataFrames* em formato Parquet: *df_sales*, que contém a informação de métricas de vendas e características temporais associadas a cada sku e *df_sku* que contém as características intrínsecas de cada sku.

Dada a complexidade das operações realizadas nesta função, a abordagem de migração adotada foi mais 'fidedigna' à versão original. Enquanto houve um estudo para otimizar o código, onde possível, a ênfase predominante foi em migrar as funcionalidades existentes de Pandas para PySpark. Na plataforma Cloudera, esta função opera de forma iterativa para cada categoria individualmente. Contudo, com a migração para Databricks, ambiciona-se um processamento simultâneo de todas as categorias. Portanto, num primeiro momento, priorizou-se a migração do código que operava com uma única categoria. Após um processo de validação dos dados, e uma vez constatada a equivalência dos *output's*, avançou-se para a adaptação do código para processamento em massa, dando sequência a novas validações.

Na infraestrutura do Cloudera, cada categoria dispunha de uma pasta específica, onde os *DataFrames df_sales* e *df_sku* eram armazenados individualmente. Contudo, com a transição para um processamento em massa, tornou-se necessário acrescentar colunas que identificassem as categorias. Esta decisão acaba por reduzir o excesso de ficheiros presentes na plataforma, mas também centralizou a informação em apenas dois *DataFrames*, facilitando, assim o acesso e a gestão dos dados. No código seguinte 4.2, encontra-se um exemplo de como criar uma nova coluna, chamada de *vendas_acumulativas* em Pandas e em PySpark. Em Pandas utiliza-se uma combinação de *groupby()* e *transform()*, aplicando o método *cumsum()* para calcular a soma cumulativa da coluna *qty_total*. Por outro lado, em PySpark é necessário utilizar uma *Window*[*]; usa-se o *partitionBy()* para segmentar os dados por sku e usa-se o *orderBy()* para ordenar os dados pela coluna *time_key*; já o método *.rowsBetween(Window.unboundedPreceding, 0)* define o alcance da janela.

[*]Window function ou funções de janela, nos sistemas de processamento de dados como o PySpark, permitem realizar cálculos num conjunto específico de linhas, sem a necessidade de agrupar todo o conjunto de dados

```
#Pandas:
df['vendas_acumulativas'] = df[['sku', 'qty_total']].groupby('sku').
    transform(lambda x: x.cumsum())

#PySpark:
window_vendas_acumulativas = Window.partitionBy('sku').orderBy('time_key').
    rowsBetween(Window.unboundedPreceding, 0)
df = df.withColumn('vendas_acumulativas', F.sum('qty_total').over(
    window_vendas_acumulativas))
```

CODE LISTING 4.2: Exemplo de código em Pandas VS em PySpark

A última função, denominada de *data_atributes*, começa por importar os dois ficheiros exportados da função anterior (*df_sales* e *df_sku*), aplicando filtros.

Começa por estabelecer um limite para determinar se um artigo alcançou 80% da sua venda dentro de períodos específicos de tempo: 13, 26 ou 52 semanas, ajudando a categorizar a duração do ciclo de vida de um produto como *Promo, Limitado 6M, Limitado 12M* ou *Contínuo*.

Faz uma análise ABC, classificando os produtos como A, B ou C dependendo da sua contribuição acumulativa para as vendas totais. Neste caso, a categoria A contém os artigos mais valiosos, contribuindo com 80% do valor total, a categoria B representa 15% e a categoria C representa os 5% menos valiosos.

Adicionalmente, a função também realiza uma classificação Syntetos-Boylan, que avalia a variabilidade e a frequência da procura dos produtos. Os produtos são classificados como *Erratic, Lumpy, Smooth* ou *Intermittent* com base na combinação de alta ou baixa variabilidade e alta ou baixa frequência.

Por fim, a função realiza uma análise sazonal para identificar a tendência de vendas dos produtos em diferentes estações, como Primavera/Verão e Outono/Inverno, bem como eventos específicos tal como Natal e Páscoa, avaliando, também, se um produto tem vendas elevadas num mês específico.

No final, a função exporta um *DataFrame* chamado de *df_sku_cat* com as informações destas análises para cada produto.

O processo de migração desta função é muito similar ao adotado para a função anterior. O objetivo central é garantir uma migração precisa e fiel. Iniciou-se a migração com foco em apenas uma categoria, permitindo validar e aperfeiçoar o código, para mais tarde alterá-lo e permitir o processamento em massa. Este método não só assegura a integridade

dos dados como também permite identificar e corrigir possíveis desafios da migração de forma mais controlada.

No código 4.3, mostra-se um exemplo ilustrativo da migração de uma operação executada em Pandas e em PySpark. Na versão Pandas, é criado um novo *DataFrame* a partir de *df_sku*. Neste é criado uma nova coluna *cumsum*, resultando de uma ordenação dos dados pelo valor da coluna *mean* em ordem decrescente. Já em PySpark, é necessário usar uma função janela, criando uma janela de dados onde se pode executar funções de agregação num subconjunto de dados. A necessidade do *partitionBy("cat_cd")* surge quando se realiza a migração para processamento em massa. No contexto iterativo anterior, todos os produtos pertenciam à mesma categoria, e por isso, não era necessário particionar os dados. Contudo, ao processar múltiplas categorias simultaneamente, é necessário garantir que a soma cumulativa é realizada apenas para produtos da mesma categoria, daí a inclusão do *partitionBy("cat_cd")*. Esta adição assegura que os resultados se mantenham consistentes com o método original em Pandas.

```
01 | #Pandas:
02 | df_cumsum = df_sku.set_index('sku').sort_values(by='mean', axis=0,
       ascending=False)['mean'].cumsum().to_frame().rename(columns={'mean': '
       cumsum'})
03 |
04 | #PySpark:
05 | window = Window.partitionBy("cat_cd").orderBy(F.desc("mean"), "sku")
06 | df_cumsum = df_sku.withColumn("cumsum", F.round(F.sum("mean").over(window),
        6))
```

CODE LISTING 4.3: Exemplo de código em Pandas VS em PySpark

Capítulo 5

Avaliação Comparativa de Desempenho: CML vs DB

A migração de algoritmos entre plataformas é uma tarefa complexa e crucial, pois pode afetar significativamente a eficácia do projeto. Neste capítulo, intitulado de "Avaliação Comparativa de Desempenho: CML vs DB", será explorado o desempenho da migração na plataforma Databricks. A ênfase recairá sobre a avaliação e comparação dos tempos de execução entre as duas plataformas, sendo feita menção às estratégias adotadas para otimizar ainda mais alguns tempos de execução em Datrabricks. É importante referir que, embora o título sugira uma comparação direta, a análise é meramente indicativa. Uma comparação direta não seria justa dado que as plataformas são distintas e os *clusters* têm recursos diferentes. Desta forma, a comparação visa proporcionar uma indicação geral das diferenças de desempenho, em vez de um confronto direto e absoluto.

5.1 Métodos de comparação

Antes de discutir a avaliação da migração, é crucial fornecer contexto sobre a base de dados. Ambas as plataformas, seja a Cloudera ou o Databricks, acedem à mesma base de dados, a qual está armazenada no Azure Data Lake Storage Gen 2 (ADLS Gen 2). As tabelas geradas durante o processo REACT estão armazenadas num *container* do ADLS Gen 2. Contudo, as tabelas oriundas do CML são armazenadas num diretório '.../DATA/nome_da_tabela', enquanto que as tabelas resultantes do processo de migração estão alocadas ao diretório '.../DATA/DB/nome_da_tabela'.

Nas tarefas que envolvem a criação de novas tabelas, além de ser relevante comparar o tempo de execução, torna-se ainda mais crítico assegurar a concordância dos dados. Durante o processo de migração, para assegurar a qualidade e garantir a coerência dos resultados entre as duas plataformas, foram estabelecidos dois testes comparativos:

1. Comparação do número de linhas entre as tabelas produzidas em CML e em Databricks;

2. Comparação do conteúdo das tabelas criadas.

A primeira comparação é efetuada conforme ilustrado na Figura 5.1. Se esta análise revelar discrepâncias, é necessário rever o código para identificar a origem do problema.

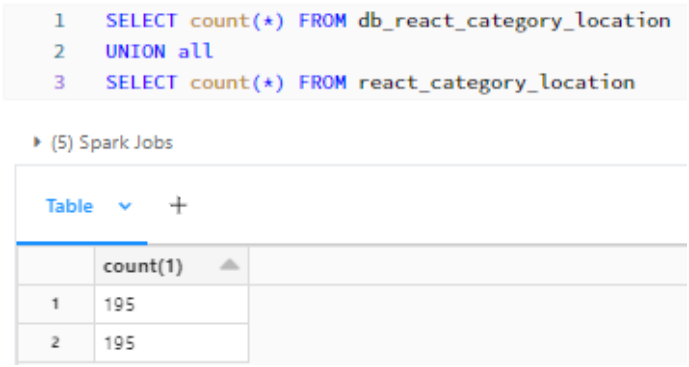


FIGURA 5.1: Análise de comparação do número de linhas entre as duas tabelas

A segunda comparação, detalhada na Figura 5.2, é mais complexa. Se esta análise resultar em diferenças, pode indicar diversas situações: existência de linhas numa tabela que não se encontram na outra ou variações em algum (ou alguns) campos das tabelas, por exemplo, se um campo apresentar o valor de 10.5 numa tabela e na outra ser 10.6. Em qualquer um destes cenário, é essencial investigar a causa do problema.

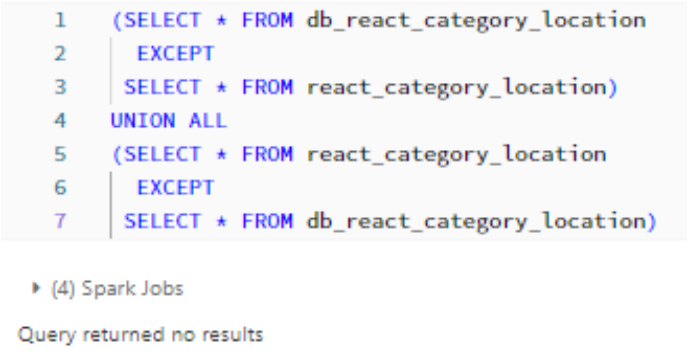


FIGURA 5.2: Análise de comparação do conteúdo entre as duas tabelas

5.2 [Hadoop] Check Data

Como referido na Secção 4.3.2, na migração desta tarefa não se realizou grandes alterações, apenas houve a adaptação de funções auxiliares, que em CML usavam bibliotecas como *ibis.impala* e *impala.dbapi*. Estas eram responsáveis por estabelecer conexões com o *cluster* Impala e executar consultas para recuperar as datas necessárias para a *pipeline*. Durante a realização modificaram-se essas funções para executarem consultas recorrendo ao mecanismo *spark.sql()*, aproveitando a capacidade de consulta SQL do ambiente Databricks, permitindo uma integração mais eficiente.

Relativamente a tempos de execução, na plataforma Cloudera a conclusão desta tarefa é realizada num intervalo médio de 1 a 2 minutos. Em Databricks, observou-se que a execução é substancialmente semelhante aos tempos de execução previamente observados.

5.3 [Hadoop] Run Before Models

Nesta tarefa existiu uma maior preocupação de otimização do código, visto que o tempo de execução total desta etapa, nas últimas execuções em Cloudera, era em média entre 6 horas e 7 horas. Como ao longo desta tarefa são executadas 21 *queries* e criadas 12 tabelas, ir-se-á, primeiro mostrar na Tabela 5.1 a comparação dos tempos de execução em CML e Databricks e seguidamente falar um pouco de certos problemas que se encontrou e como é que estes foram resolvidos.

Ao analisar-se a Tabela 5.1, evidencia-se uma significativa melhoria no tempo de execução para determinadas tabelas específicas e no aumento do tempo de execução em duas tabelas. De seguida, abordar-se-ão as otimizações implementadas que contribuíram para algumas melhorias no tempo de execução:

- **Tabela react_master**: Em CML, esta tabela processava múltiplas categorias de forma sequencial. Isto significa que a consulta era executada para uma categoria de cada vez, sendo posteriormente a mesma inserida na tabela, para todas as categorias. Durante o processo de migração, optou-se por uma abordagem de processamento em massa das categorias. Consequentemente, a consulta foi reestruturada para que pudesse processar simultaneamente todas as categorias, o que também acontece para outras tabelas, como por exemplo a **tabela react_master_warehouse**;

- **Tabela react_s_prod**: Em CML, esta tabela estava a ser povoada por três consultas distintas. Após uma análise cuidadosa das interações e dependências entre as consultas, identificou-se uma oportunidade de otimização, levando à combinação destas três consultas numa única, mais eficiente e simplificada, o que também ocorreu na tabela **tabela react_models**;

- Um dos métodos aplicados de otimização foi a utilização de ***broadcast*** (referenciada na Secção 2.6.3). O *broadcasting*, refere-se à ação de enviar um *DataFrame* pequeno para todos os nós de um *cluster*, de forma a otimizar a união com um DF significamente maior;

- Redução do uso de *Common Table Expression* (CTE). Os CTEs são frequentemente usados para segmentar uma consulta complexa, tornando-a mais legível e organizada. Contudo, o uso excessivo de CTEs pode, em algumas situações, aumentar o tempo de execução, visto que cada CTE é lido e armazenado em memória. A **tabela react_sku_week** continha 12 CTEs distintos. Ao reduzir este número, a consulta pode beneficiar de otimizações de plano de execução realizados pelo Spark, uma vez que, em vez de avaliar múltiplas expressões intermédias, o Spark otimiza a consulta como um todo, evitando a materialização desnecessário de resultados intermédios;

- Na tabela **react_sku_week**, existe uma Secção específica do código, onde ocorre a união de quatro tabelas distintas. Reorganizou-se a ordem da união de forma a que se começasse com a tabela com mais registos e que se acabasse com a tabela com menos registos. Ao começar com uma tabela maior, o universo de linhas é definido desde o início, potencialmente reduzindo a quantidade de dados que precisam de ser transferido entre os nós durante as seguintes etapas. O que ajudou nesta eficiência foi que a tabela maior tinha várias *join keys*, isto ajuda a distribuir o trabalho de forma mais uniforme entre os nós e pode reduzir a quantidade de dados transferidos durante as uniões.

- Inicialmente, a tabela **react_sku_week** apresentava tempos de execução a rondar 5 horas, mas ao realizar as últimas duas otimizações referidas, conseguiu-se reduzir esse tempo para 3 horas.

TABELA 5.1: Comparação dos tempos finais de execução CML vs DBs

Tabelas	Cloudera	Databricks
react_product	1 - 2 minutos	4 - 5 minutos
react_location	1 - 2 minutos	1 - 2 minutos
react_cat_location	2 minutos	2 minutos
react_time	1 - 2 minutos	1 - 2 minutos
react_master	4 horas	15 - 20 minutos
react_warehouse	5 minutos	2 minutos
react_s_prod_retek	15 - 20 minutos	15 - 20 minutos
react_s_prod	10 minutos	1 - 2 minutos
react_season_dim	1 - 2 minutos	3 - 4 minutos
react_udas	8 minutos	1 - 2 minutos
react_sku_promo	5 minutos	Incorporou-se na react_models
react_sku_week	1 hora 30 minutos	3 horas
react_models	45 minutos	15 - 20 minutos
Total	6 - 7 horas	4 horas

É importante destacar que, apesar da melhoria de duas horas no tempo de execução desta etapa, o principal *gargalo* continua a ser a tabela react_sku_week. Conforme ilustrado nas Figuras 4.8 e 4.9, a tabela final, react_models, depende da conclusão da primeira para inicial o seu processamento. A otimização deste código não se encontra completa, ficando para segundo plano, para haver uma discussão e um planeamento ainda melhor.

5.4 [CV] Run Data Processing

Nesta fase, conforme mencionado na Secção 4.3.4, o processo é composto por três funções distintas. A primeira função foi integrada na estrutura lógica da tabela react_models, não havendo impacto no tempo de criação da tabela. Em outras palavras, a tabela foi inicialmente criada e os tempos de execução foram registados. Posteriormente, a *query* foi adaptada para incorporar a nova lógica, resultando em tempos de execução semelhantes aos registados anteriormente. Sendo que, na plataforma CML, esta função apresentava um tempo médio de execução que varia entre os 25 e os 30 minutos.

Por outro lado, a segunda função, denominada de *run_data_processing*, em CML, apresenta um tempo médio de execução que varia entre os 20 e os 25 minutos. Inicialmente, foi

realizada uma migração meticulosa, quase linha por linha, o que resultou num tempo de execução de 45 a 55 minutos, podendo ser atribuídos vários motivos, como por exemplo, a não otimização do código para PySpark e para Databricks, a transcrição de operações redundantes, entre outros motivos.

No entanto, após executar o código migrado, identificaram-se diversas partes do código que poderiam ser otimizadas. Um exemplo é a otimização das operações de união (ou *join* em PySpark) da tabela com ela própria. Estas operações podem ser reformuladas para alcançar um desempenho mais eficiente. Estas operações são consideradas caras devido à natureza distribuída e paralela do Spark. O Spark realiza o processamento num *cluster*, onde distribui e processa os dados em várias partições e nós. Quando uma operação *join* ou *merge* ocorre, várias partições de dados precisam de ser unidas e ordenadas, exigindo coordenação entre os diferentes nós do *cluster*, resultando numa sobrecarga no processamento. Outra otimização significativa envolveu o uso de *cache* para armazenar em memória *DataFrames* frequentemente usados, reduzindo a necessidade de leituras repetidas do disco, que é uma comparação lenta relativamente à leitura em memória.

Após as melhorias realizadas a esta função, o tempo de execução em Databricks passou de 45 a 55 minutos para entre os 30 e 35 minutos. Apesar da melhoria já realizada, continuar-se-á a explorar mais características e recursos do Databricks e do Spark para alcançar uma otimização ainda melhor.

No excerto de código apresentado em 5.1, é exemplificado como criar uma coluna de média de vendas ao realizar-se um *left join* à mesma tabela. Na abordagem seguinte recorre-se à funcionalidade *window* do PySpark para calcular essa média de forma mais eficiente.

```
01 | #Antes de otimizar:
02 | df = df.join(df.groupBy("sku").agg({"sales": "avg"}), on="sku", how="left")
03 |
04 | #Depois de otimizar:
05 | window = Window.partitionBy("sku")
06 | df = df.withColumn("avg_sales", F.avg("sales").over(window))
```

CODE LISTING 5.1: Exemplo de otimização de um left join

A última função a ser executada nesta tarefa, denominada de *data_attributes*, em Cloudera, apresenta um tempo médio de execução de 5 minutos, que coincide com o tempo médio de execução observado em Databricks.

Capítulo 6

Conclusão e Trabalho Futuro

Este capítulo final será divido em duas Secções distintas. O primeiro será dedicado à "Conclusão", onde será apresentado um resumo dos principais pontos e resultados obtidos ao longo deste trabalho, juntamente com algumas lições aprendidas. A segunda parte falará do "Trabalho Futuro", onde serão exploradas possíveis direções que podem ser seguidas para melhorar o código já migrado e onde será traçado um plano para o restante algoritmo.

6.1 Conclusão

Durante o desenvolvimento deste trabalho, explorou-se tecnologias e infraestruturas de análise de dados, bem como a inicialização da migração do algoritmo REACT para a plataforma Databricks. A migração para Databricks demonstrou benefícios de utilizar as últimas tecnologias emergentes. Além de modernizar a infraestrutura subjacente do algoritmo, também abriu caminho para análises e implementações mais escaláveis e eficientes.

No entanto, ao longo do processo de migração, foram-se encontrando alguns desafios significativos que se revelaram oportunidades valiosas de aprendizagem. É essencial destacar que migrações desta natureza exigem uma planificação cuidadosa, juntamente com um profundo entendimento das arquiteturas tanto de origem quanto de destino. Além disso, um estudo aprofundado e abrangente do próprio algoritmo revelou pontos possíveis de melhorias substanciais. A experiência ao longo deste trabalho também mostrou a importância de migrar a lógica do algoritmo (ou de uma tarefa específica) como um todo,

em vez de realizar migração linha por linha. A migração da lógica mostra-se fundamental para evitar ineficiências no código, tornando-se numa análise pré-migração valiosa.

É importante ressaltar que existe sempre espaço para uma otimização contínua. Mesmo com tempos de execução aceitáveis, sabe-se que há espaço para melhorias, especialmente à medida que se continua a aprofundar o conhecimento na plataforma Databricks e em Spark.

Resumindo, a migração do algoritmo REACT da plataforma Cloudera Machine Learning para a plataforma Databricks mostra uma modernização na infraestrutura da Sonae, abrindo caminhos para análises mais escaláveis e eficazes, alinhando-se com as necessidades de um mundo cada vez mais moderno.

6.2 Trabalho Futuro

Este projeto representa o estágio inicial de uma migração contínua e de um processo de uma jornada contínua de aprimoramento e otimização do código. Neste subcapítulo, serão delineadas algumas áreas-chave que merecem atenção no âmbito do trabalho futuro.

Conforme mencionado anteriormente, embora se tenha alcançado resultados satisfatórios na fase inicial da migração, reconhece-se que existe sempre espaço para a otimização de código. Pretende-se realizar uma análise mais aprofundada das funcionalidades do Spark e da plataforma Databricks, com o intuito de explorar ao máximo as suas capacidades. Este esforço será acompanhado por uma análise rigorosa do código já migrado, com foco na melhoria da eficiência e na redução dos tempos de execução.

O processo de migração prosseguirá para as restantes partes do algoritmo, garantindo a transferência de toda a lógica do algoritmo para a plataforma Databricks, com o objetivo de deixar de depender da infraestrutura anterior, o Cloudera Machine Learning.

Um aspeto relevante a destacar neste subcapítulo é o planeamento da biblioteca Sparkling Water no contexto do modelo de *Price Elasticity of Demand*. Enquanto que em CML se utilizava a biblioteca H2O, pretende-se aproveitar as capacidades da biblioteca Sparkling Water, que proporciona uma combinação entre a rapidez e escalabilidade de algoritmos de *machine learning* com as funcionalidades do Spark, permitindo uma análise mais eficiente.

Além disso, no modelo Ciclo de Vida, planea-se paralelizar as etapas de classificação e previsão, em vez de realizá-las sequencialmente. As consultas de pós-processamento

serão tratadas de forma similar às de pré-processamento. Por fim, os restantes processos ou *pipelines* serão tratados da mesma forma.

Em conclusão, o trabalho futuro está envolvido com vários compromissos, incluindo a melhoria contínua do código e a sua inovação, maximizando o valor dos dados e mantendo a Sonae na vanguarda da análise de dados num mundo cada vez mais digital e moderno.

Apêndice A

Diagrama dos Processos

A.1 Diagramas do processo de atualização semanal de indicadores e métricas

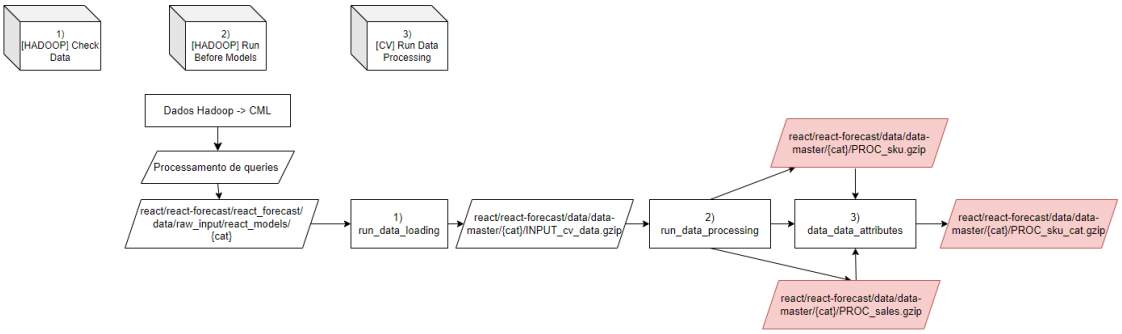


FIGURA A.1: Diagrama da parte do Pré-processamento da *pipeline* 3.5

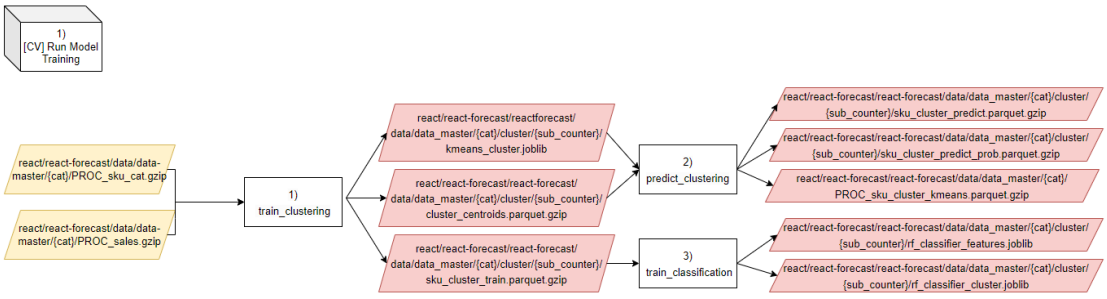


FIGURA A.2: Diagrama do fluxo de código do ramo de cima - Parte I da *pipeline* 3.5

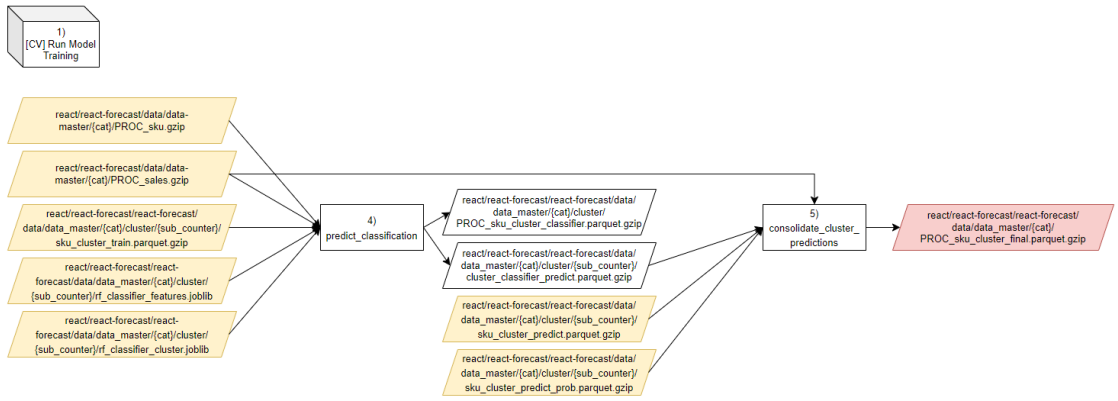


FIGURA A.3: Diagrama do fluxo de código do ramo de cima - Parte II da *pipeline* 3.5

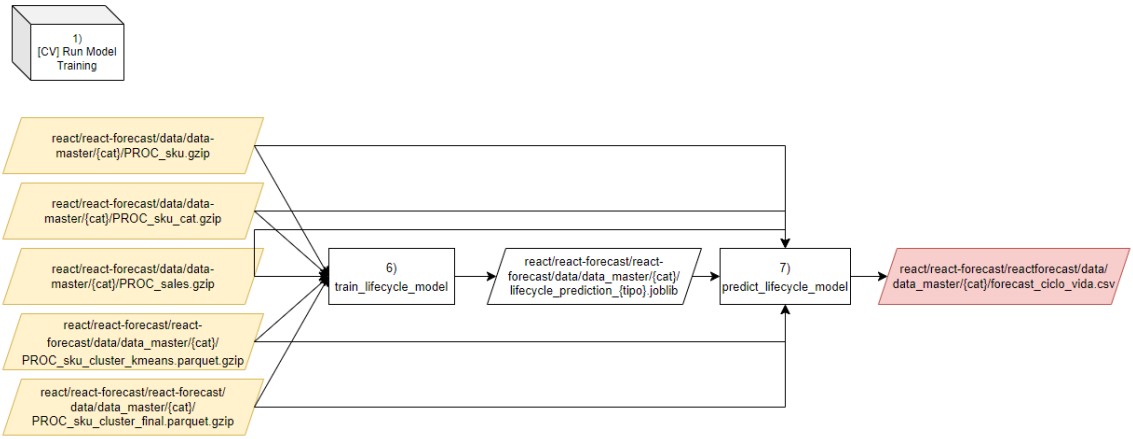


FIGURA A.4: Diagrama do fluxo de código do ramo de cima - Parte III da *pipeline* 3.5

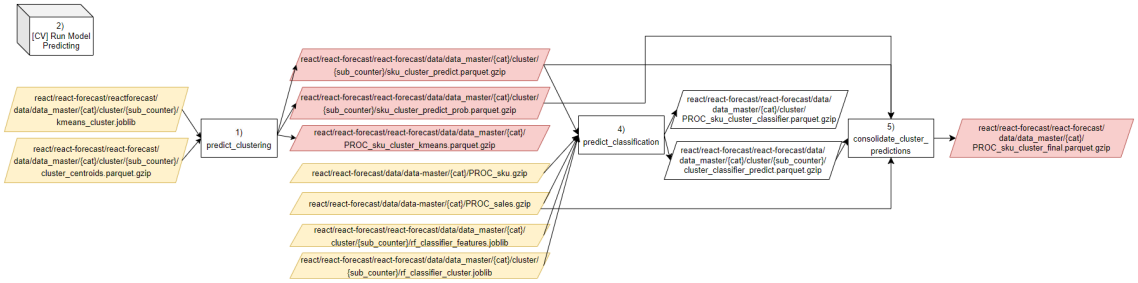


FIGURA A.5: Diagrama do fluxo de código do ramo de cima - Parte IV da *pipeline* 3.5

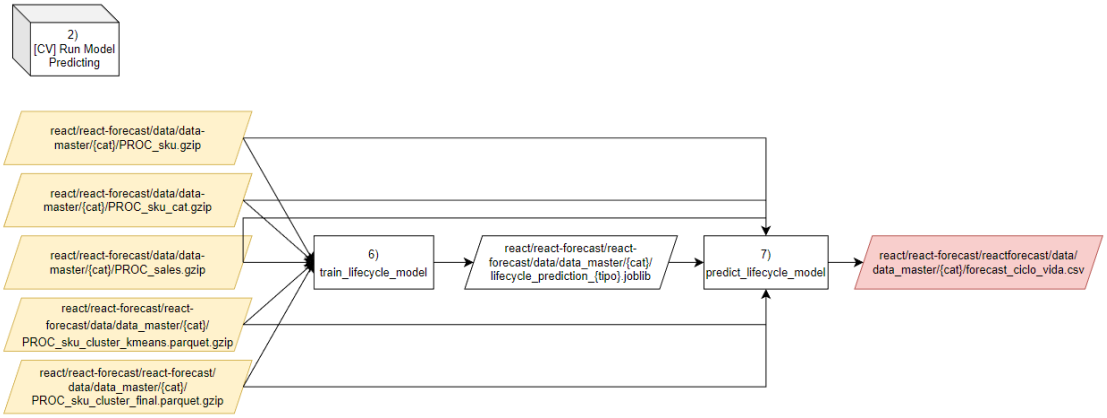


FIGURA A.6: Diagrama do fluxo de código do ramo de cima - Parte V da *pipeline* 3.5

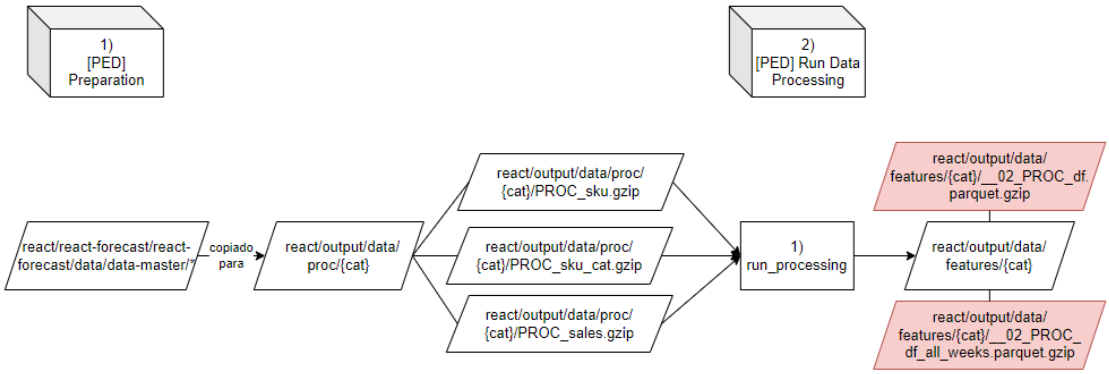


FIGURA A.7: Diagrama do fluxo de código do ramo de baixo - Parte I da *pipeline* 3.5

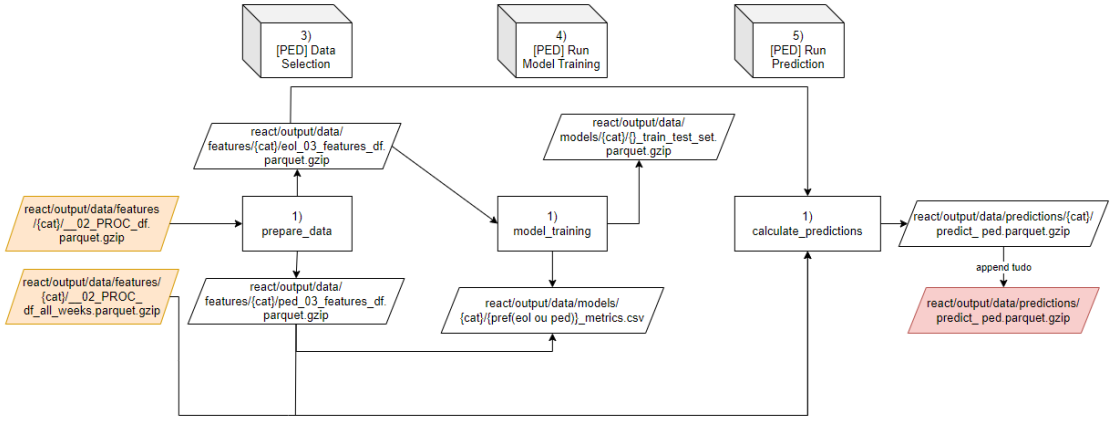


FIGURA A.8: Diagrama do fluxo de código do ramo de baixo - Parte II da *pipeline* 3.5

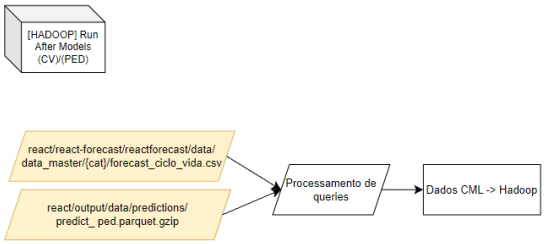


FIGURA A.9: Diagrama do fluxo de código do Pós-processamento *pipeline* 3.5

A.2 Diagramas do processo de treino do modelo de Price Elasticity of Demand

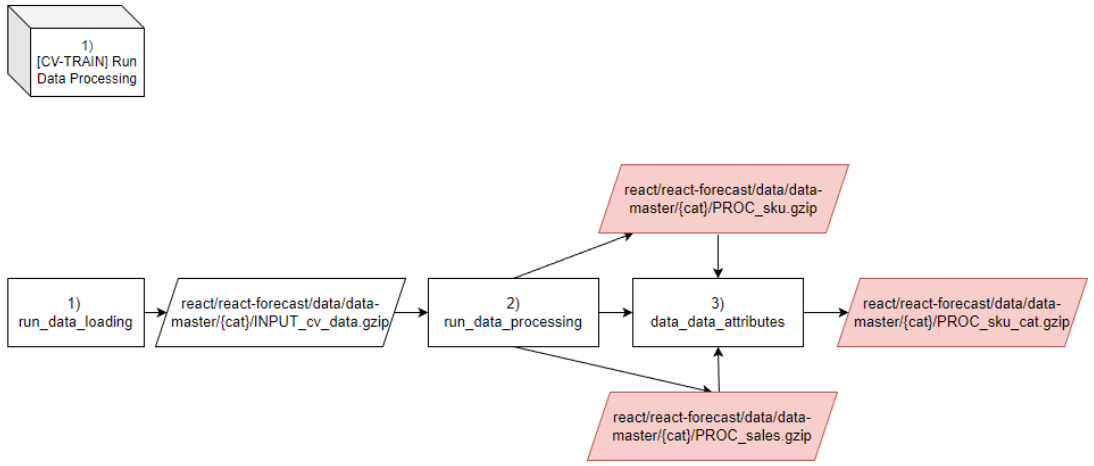


FIGURA A.10: Diagrama do fluxo de código da *pipeline* 3.6 - Parte I

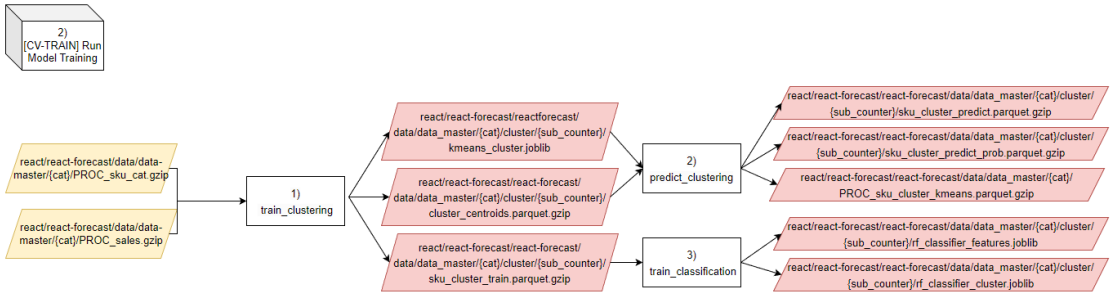


FIGURA A.11: Diagrama do fluxo de código da *pipeline* 3.6 - Parte II

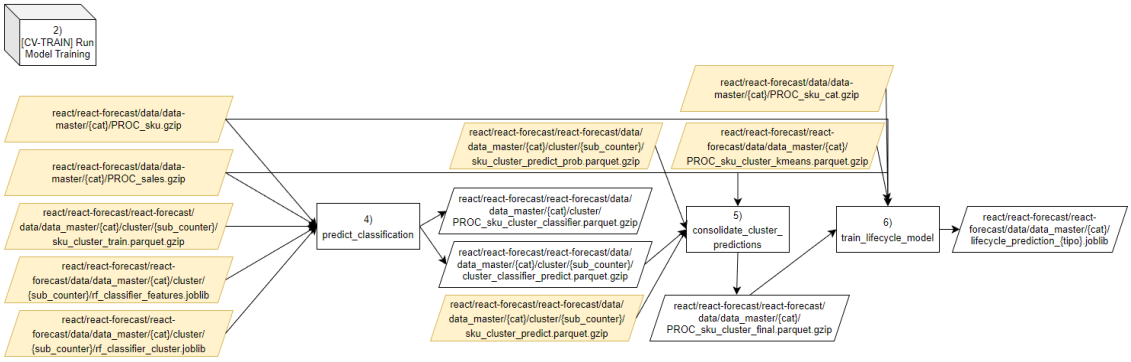


FIGURA A.12: Diagrama do fluxo de código da *pipeline* 3.6 - Parte III

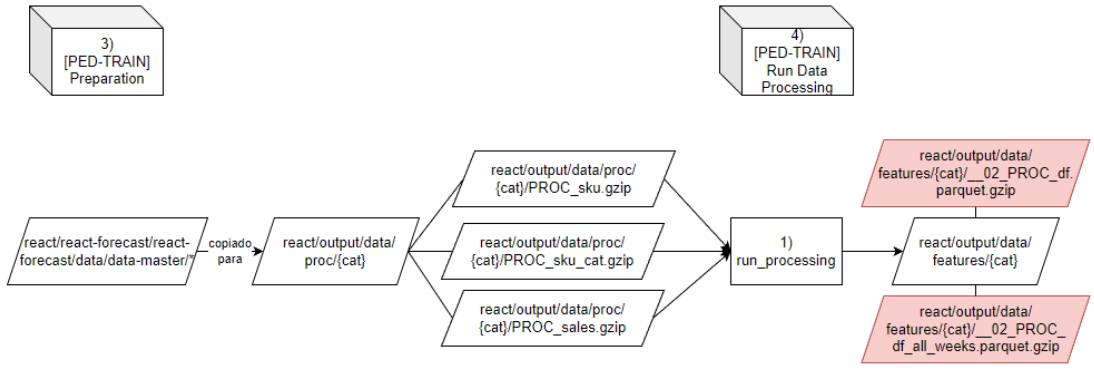


FIGURA A.13: Diagrama do fluxo de código da *pipeline* 3.6 - Parte IV

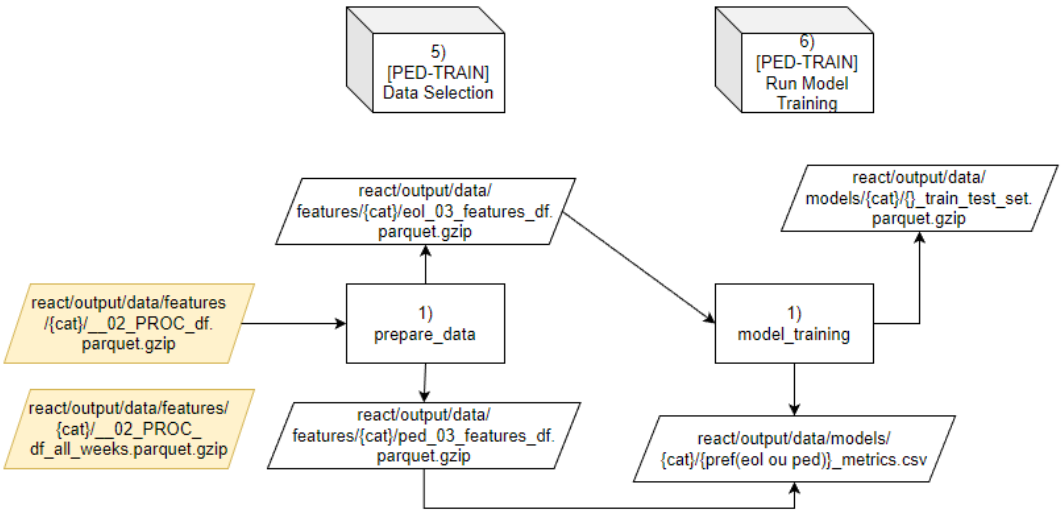


FIGURA A.14: Diagrama do fluxo de código da *pipeline* 3.6 - Parte V

A.3 Diagramas do processo de adição de novas categorias à estrutura REACT

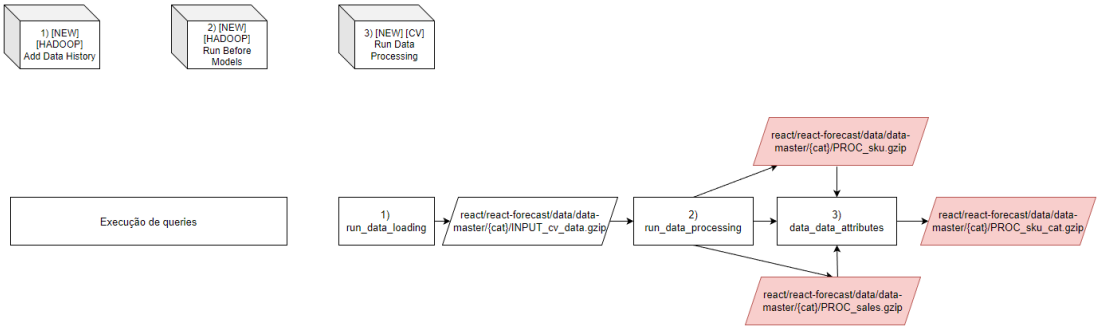


FIGURA A.15: Diagrama do fluxo de código da *pipeline* 3.7 - Parte I

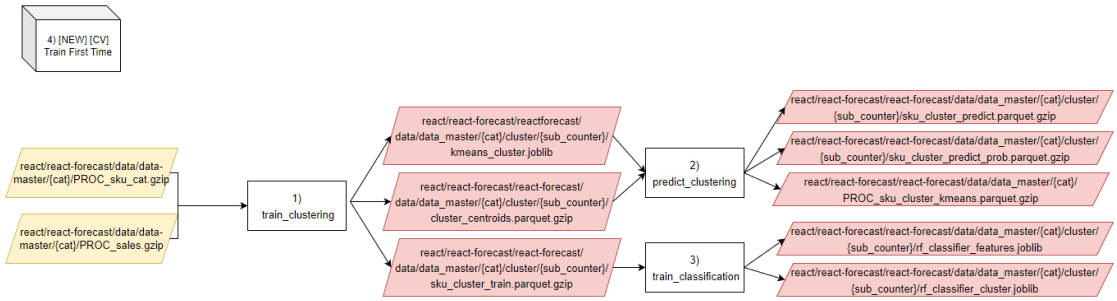


FIGURA A.16: Diagrama do fluxo de código da *pipeline* 3.7 - Parte II

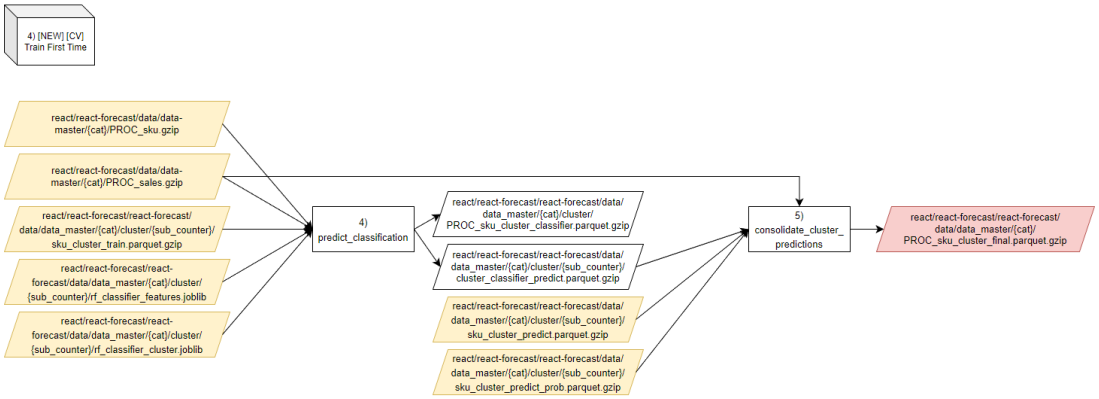


FIGURA A.17: Diagrama do fluxo de código da *pipeline* 3.7 - Parte III

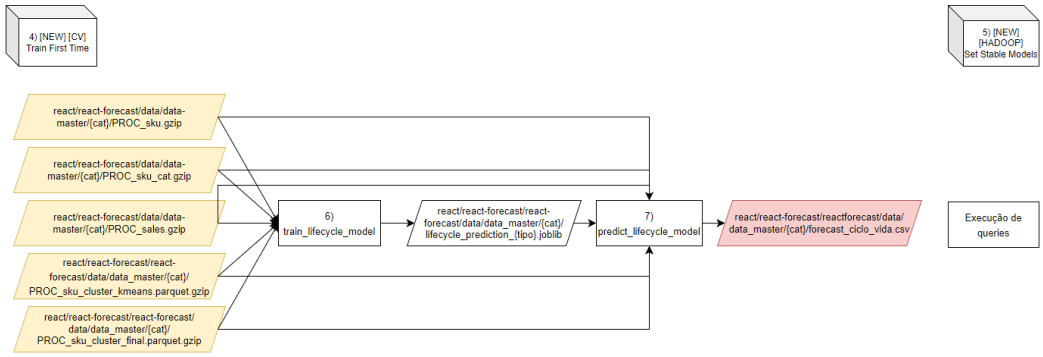


FIGURA A.18: Diagrama do fluxo de código da *pipeline* 3.7 - Parte IV

A.4 Diagramas do processo de avaliação dos resultados de uma categoria

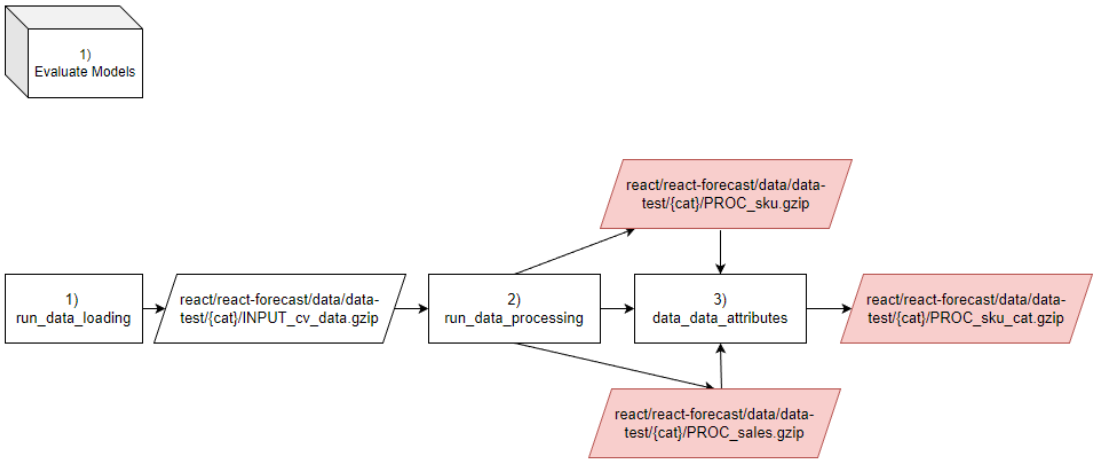


FIGURA A.19: Diagrama do fluxo de código da *pipeline* 3.8 - Parte I

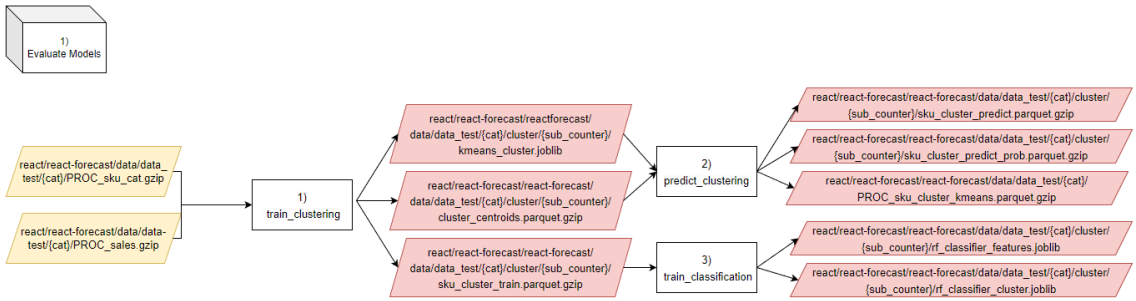


FIGURA A.20: Diagrama do fluxo de código da *pipeline* 3.8 - Parte II

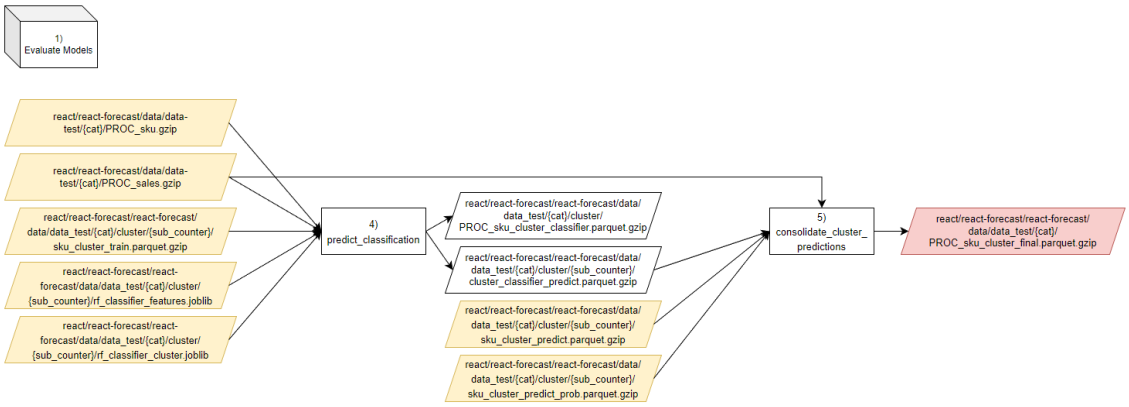


FIGURA A.21: Diagrama do fluxo de código da *pipeline* 3.8 - Parte III

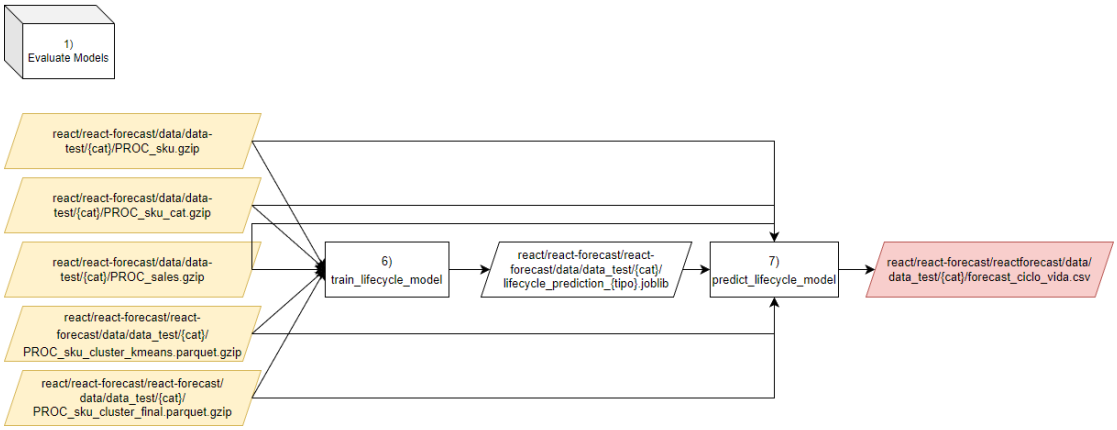


FIGURA A.22: Diagrama do fluxo de código da *pipeline* 3.8 - Parte IV

A.5 Diagrama das *queries* do pré-processamento de dados

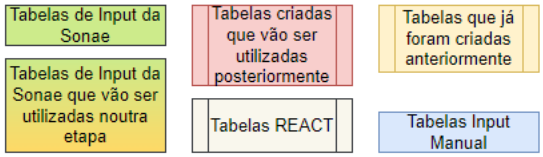


FIGURA A.23: Legenda dos diagramas das *queries*

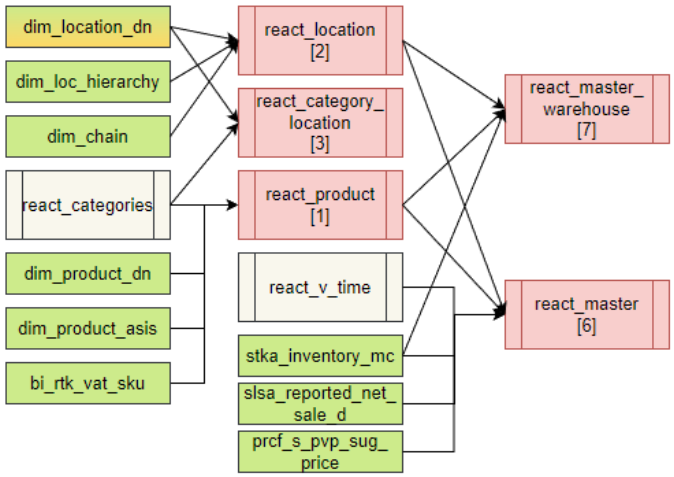


FIGURA A.24: Diagrama do fluxo das *queries* 3.5 - Parte I

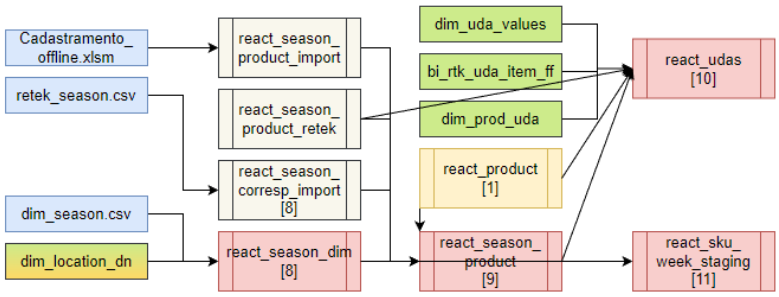


FIGURA A.25: Diagrama do fluxo das *queries* - Parte II

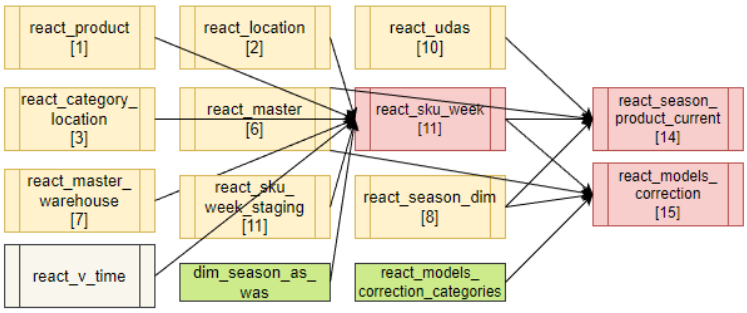


FIGURA A.26: Diagrama do fluxo das *queries* - Parte III

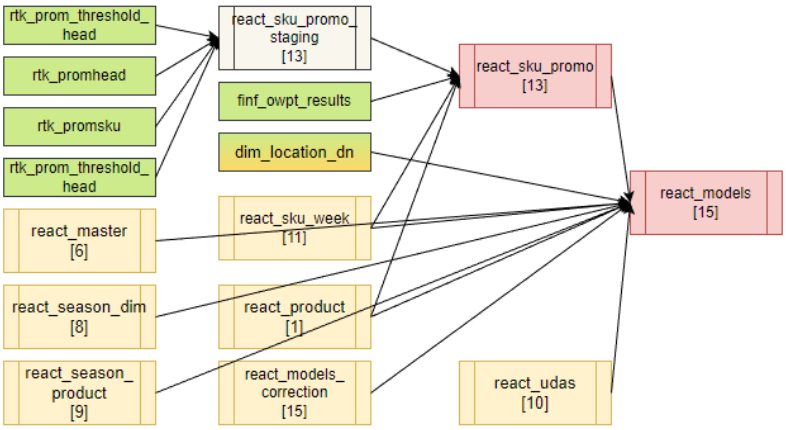


FIGURA A.27: Diagrama do fluxo das *queries* - Parte IV

Bibliografia

Apache Hadoop Yarn [Accessed on 19-12-2022]. (s.d.). https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html

Armbrust, M., Fan, W., Xin, R., & Zaharia, M. (2016). Introducing Apache Spark Datasets [Accessed on 20-12-2022]. https://www.databricks.com/blog/2016/01/04/introducing-apache-spark-datasets.html

Cukier, K., & Mayer-Schoenberger, V. (2013). The rise of big data: How it's changing the way we think about the world. *Foreign Aff.*, *92*, 28.

Damji, J., & Lee, D. (s.d.). Apache Spark Key Terms, Explained [Accessed on 20-12-2022]. https://www.databricks.com/blog/2016/06/22/apache-spark-key-terms-explained.html

Databricks. (s.d.-a). Adaptive query execution - Azure Databricks [Accessed on 20-12-2022]. https://learn.microsoft.com/en-us/azure/databricks/optimizations/aqe#dynamically-change-sort-merge-join-into-broadcast-hash-join

Databricks. (s.d.-b). Delta Table Properties Reference - Azure Databricks [Accessed on 20-12-2022]. https://learn.microsoft.com/en-us/azure/databricks/delta/table-properties

Databricks. (s.d.-c). Introduction to data lakes. https://www.databricks.com/discover/data-lakes

Databricks. (s.d.-d). Optimize performance with caching on databricks [Accessed on 20-12-2022]. https://docs.databricks.com/optimizations/disk-cache.html

Databricks. (s.d.-e). What are all the delta things in azure databricks? - azure databricks [Accessed on 20-12-2022]. https://learn.microsoft.com/en-us/azure/databricks/introduction/delta-comparison

Databricks. (s.d.-f). What is Delta Lake? [Accessed on 20-12-2022]. https://docs.databricks.com/delta/index.html

Databricks. (s.d.-g). What is Delta Lake? [Accessed on 20-12-2022]. https://docs.databricks.com/delta/index.html

Databricks. (s.d.-h). Widgets do databricks – Azure Databricks [Accessed on 20-12-2022]. https://learn.microsoft.com/pt-pt/azure/databricks/notebooks/widgets

Databricks. (2020). *What are spark applications?* [Accessed on 26-10-2022]. https://www.databricks.com/glossary/what-are-spark-applications

Databricks. (2022). *What is pyspark?* [Accessed on 19-12-2022]. https://www.databricks.com/glossary/pyspark

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 137–150.

Different phases of map reduce in word count example [Accessed on 05-01-2023]. (s.d.). https://www.researchgate.net/figure/The-Overall-MapReduce-Word-Count-Process_fig1_326812621

Forbes. (2019). Council post: 12 tips to smoothly migrate your company's team to a new software platform [Accessed on 10-04-2023]. https://www.forbes.com/sites/forbestechcouncil/2019/04/03/12-tips-to-smoothly-migrate-your-companys-team-to-a-new-software-platform/?sh=50f5807c7753

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The Google File System. *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 20–43.

Gupta, D., & Rani, R. (2019). A study of big data evolution and research challenges. *Journal of information science*, *45*(3), 322–340.

Kala Karun, A., & Chitharanjan, K. (2013). A review on hadoop — HDFS infrastructure extensions. *2013 IEEE Conference on Information Communication Technologies*, 132–137. https://doi.org/10.1109/CICT.2013.6558077

Laney, D. (2001). *3D Data Management: Controlling Data Volume, Velocity, and Variety* (rel. téc.). META Group. http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf

Linkedin. (s.d.). What are the best practices for migrating your content to a new platform? [Accessed on 10-04-2023]. https://www.linkedin.com/advice/0/what-best-practices-migrating-your-content-new

Maposa, T. T., & Sethi, M. (2018). *SQL-on-Hadoop: The Most Probable Future in Big Data Analytics*.

Merceedi, K. J., & Sabry, N. A. (2021). A Comprehensive Survey for Hadoop Distributed File System. *Asian Journal of Research in Computer Science*.

Microsoft. (s.d.-a). Azure Data Lake Storage Gen2 introduction - azure storage [Accessed on 25-09-2023]. https://learn.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-introduction

Microsoft. (s.d.-b). *Introduction to azure data factory - azure data factory* [Accessed on 05-01-2023]. https://learn.microsoft.com/en-us/azure/data-factory/introduction

Microsoft. (s.d.-c). *What is Azure-Microsoft Cloud Services: Microsoft Azure* [Accessed on 05-01-2023]. https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-azure/

Palmucci, J. (s.d.). *Engineering/product/operations a Tripadvisor blog* [Accessed on 20-12-2022]. https://www.tripadvisor.com/engineering/using-apache-spark-for-massively-parallel-nlp/

Palmucci, J. (2015). Engineering/product/operations a Tripadvisor blog [Accessed on 20-12-2022]. https://www.tripadvisor.com/engineering/using-apache-spark-for-massively-parallel-nlp/

Qin, X., Chen, Y., Chen, J., Li, S., Liu, J., & Zhang, H. (2017). *The Performance of SQL-on-Hadoop Systems - An Experimental Study*. https://doi.org/10.1109/BigDataCongress.2017.68

Salloum, S., Dautov, R., Chen, X., Peng, P. X., & Huang, J. Z. (2016). Big data analytics on Apache Spark. *International Journal of Data Science and Analytics*, *1*, 145–164.

Sonae. (s.d.-a). *História Sonae* [Accessed on 26-10-2022]. https://www.sonae.pt/pt/sonae/historia/

Sonae. (s.d.-b). *Marcas Sonae* [Accessed on 26-10-2022]. https://www.sonae.pt/pt/sonae/marcas/

Sonae. (s.d.-c). *Sonae no Mundo* [Accessed on 26-10-2022]. https://www.sonae.pt/pt/sonae/onde-estamos/

Spark, A. (s.d.-a). Cluster Mode Overview [Accessed on 19-12-2022]. https://spark.apache.org/docs/latest/cluster-overview.html

Spark, A. (s.d.-b). *MLlib: RDD-based API - Spark 3.3.1 Documentation.* [Accessed on 19-12-2022]. https://spark.apache.org/docs/latest/mllib-guide.html

Spark, A. (s.d.-c). *RDD Programming Guide* [Accessed on 2022-11-30]. https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

Stančin, I., & Jović, A. (2019). An overview and comparison of free Python libraries for data mining and big data analysis. *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics*, 977–982. https://doi.org/10.23919/MIPRO.2019.8757088

Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. (2013). Apache hadoop yarn: Yet another resource negotiator. *Proceedings of the 4th annual Symposium on Cloud Computing*, 1–16.

White, T. (2012). *Hadoop: The definitive guide*. "O'Reilly Media, Inc."

Xin, R., Armbrust, M., & Lu, D. (2015). Introducing DataFrames in Apache Spark for Large Scale Data Science [Accessed on 20-12-2022]. https://www.databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html

Xin, R. S., Crankshaw, D., Dave, A., Gonzalez, J. E., Franklin, M. J., & Stoica, I. (2014). *GraphX: Unifying Data-Parallel and Graph-Parallel Analytics*. arXiv: 1402.2394. http://arxiv.org/abs/1402.2394

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S., & Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 15–28.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., Stoica, I., et al. (2010). Spark: Cluster computing with working sets. *HotCloud*, *10*(10-10), 95.

Zaman, A. U. (2023). Pandas vs pyspark..! [Accessed on 20-12-2022]. https://medium.com/geekculture/pandas-vs-pyspark-fe110c266e5c