INGESTÃO DE DADOS EM UM AMBIENTE CLOUD

Lucas Schafer Vrielink[1], **William Moraes da Silva**[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul (IFRS)
Campus Farroupilha - RS - Farroupilha
lucaasvrielink@gmail.com, william.silva@farroupilha.ifrs.edu.br

Resumo. Ambientes de Big Data apresentam desorganização nas estruturas de ingestão de dados comprometendo a eficiência, escalabilidade e confiabilidade dos processos analíticos. Diante disso, este trabalho propôs a construção de uma ingestão em lotes utilizando o Databricks na nuvem Azure com padronização das fontes de dados por meio do Java Database Connectivity (JDBC), definição de métodos de atualização e avaliação de desempenho do método aplicado. Como resultado, obteve-se uma ingestão com bom desempenho e com dados íntegros por meio da escolha adequada do método de ingestão. Contudo, técnicas como adaptive query execution e CDC podem ser exploradas futuramente.

Abstract. Big Data environments often show disorganization in data ingestion structures, compromising the efficiency, scalability, and reliability of analytical processes. To address this, this work proposed the construction of a batch ingestion using Databricks on the Azure cloud, with standardized data sources via Java Database Connectivity (JDBC), definition of update methods, and performance evaluation of the applied approach. As a result, the ingestion achieved good performance and data integrity through the appropriate choice of ingestion method. However, techniques such as adaptive query execution and CDC can be explored in future work.

1. Introdução

A ingestão de dados é um processo fundamental em um ambiente de dados, visto que ela permite a coleta, transformação e armazenamento de informações de diversas fontes.

Com isso, viabiliza a disponibilidade dos dados para análises que auxiliam na tomada de decisões estratégicas (IBM, 2024).

No contexto de Big Data, que se refere ao grande volume de dados que não pode ser processado de forma tradicional devido à sua quantidade, variedade ou velocidade, a ingestão pode ocorrer de duas formas: lotes ou *streaming*. A ingestão em lotes envolve o processamento de grandes volumes de dados em intervalos programados, ou seja, os dados são coletados, armazenados e processados em lotes e em horários específicos. Já a ingestão em *streaming* ocorre de forma contínua e em tempo real, processando os dados à medida que eles são gerados. Ferramentas como Azure Data Factory são usadas para ingestões em lotes, enquanto Apache Kafka e Azure Event Hubs são projetadas para permitir ingestões em *streaming* (Taurion, 2013; Microsoft, 2025).

A computação em nuvem tem trazido benefícios para empresas em ambientes de dados, proporcionando escalabilidade e alta disponibilidade que facilitam o processamento de dados. Plataformas de computação em nuvem como a Azure, oferecem serviços específicos para ingestão de dados, além de integrar com outros serviços como o Databricks, um ambiente de dados que permite implementar soluções sob o *framework* Apache Spark para processar grandes volumes de dados em paralelo e aumentar a eficiência em construir *pipelines*, ou seja, processos de transformação de dados, escaláveis e automatizadas (Databricks 2025; Pedrosa, Nogueira, 2011).

Entretanto, a implementação desses *pipelines* de dados apresentam desafios consideráveis, como a necessidade de otimizar as ingestões e integrar com tecnologias emergentes ou novas fontes de dados. Para se adequar a várias fontes de dados, a ingestão deve utilizar uma forma de conexão adotada por vários bancos de dados, considerar a falta de informações sobre atualização de registros, a padronização inconsistente de colunas e tabelas, a possível ausência de uma coluna de identificação devido ao uso de chaves compostas e a necessidade de uma estratégia eficiente para evitar impactos na performance durante a extração de grandes tabelas. Além disso, estudos demonstram que arquiteturas bem planejadas podem mitigar essas dificuldades e aumentar a eficiência dos *pipelines*, tornando-as mais alinhadas às necessidades. As vantagens dessa abordagem incluem a redução de custos operacionais e a melhoria da segurança, com soluções mais robustas de controle de acesso e proteção de dados, porém com a possibilidade de criar dependências com o provedor de serviços cloud, que limita a flexibilidade das organizações (Cândido, Araújo Júnior, 2022; Rapôso et al, 2024; Lin et al, 2022; Fehad, 2019).

A motivação para este estudo surgiu da necessidade observada no meu ambiente de trabalho, onde a estrutura de ingestão de dados apresentava desafios relacionados à desorganização, dificultando a eficiência e escalabilidade dos processos. Desta forma, este projeto propõe o desenvolvimento de uma ingestão de dados utilizando as soluções baseadas em computação em nuvem, aproveitando a escalabilidade oferecida por esse modelo de serviço, além de reduzir a dependência de ambientes cloud, selecionando os que optam por ferramentas open-source e aumentando a flexibilidade no desenvolvimento. O trabalho está dividido da seguinte forma: na seção 2, é apresentada a fundamentação teórica, abordando conceitos sobre ingestão de dados e seus desafios, *data lakes* e o ambiente Databricks. Na seção 3, são apresentados os procedimentos metodológicos, definindo o método de ingestão e a sua aplicação. Na seção 4 são apresentados o desenvolvimento do algoritmo e os resultados obtidos, que são discutidos na seção 5. Por fim, na seção 6 são explanadas as considerações finais.

2. Objetivos

2.1. Objetivo Geral

Desenvolver uma ingestão de dados em ambiente cloud utilizando a plataforma Azure e o Databricks, baseado na arquitetura medalhão.

2.2. Objetivos Específicos

- Implementar um processo de ingestão de dados em lotes no Databricks.
- Organizar os dados em camadas com a arquitetura medalhão.
- Analisar o desempenho e a eficiência dos diferentes métodos de ingestão de dados, considerando aspectos como tempo de processamento e consumo de recursos computacionais.
- Implementar monitoramento da ingestão de dados utilizando *logs* no Datadog.
- Explorar possíveis otimizações no *pipeline* para reduzir custos e melhorar a performance da ingestão de dados.

3. Referencial Teórico

Para auxiliar na compreensão do estudo, esta seção foi dividida em tópicos que abordam o tema da pesquisa, trazendo informações e conceituando sobre ambientes e ingestões de dados na cloud e seus artefatos.

3.1. Processamento e Ingestão de Dados

Dados são gerados a todo momento por meio de transações online, e-mails, vídeos, imagens, sistemas organizacionais, posts nas redes sociais, sensores, etc. Essa grande quantidade de dados é armazenada sem padronização em bancos de dados massivos e, com o tempo, torna-se difícil de operar. Ao mesmo tempo, a demanda por dados de qualidade tem crescido significativamente, tornando-os cada vez mais valiosos em diversas áreas da sociedade. Contudo, apesar da disponibilidade para uso, cerca de 73% dos dados coletados das organizações não são utilizados para análises devido à falta de qualidade, ou seja, ainda precisam ser tratados para serem entendidos e se tornarem de fato úteis (Databricks, 2023; Sagiroglu, Sinac, 2013).

A ingestão de dados se baseia em trazer dados com qualidade de diversas fontes e de forma eficiente para um único repositório. Também conhecidos como *ETL* (*Extract, Transform and Load* - Extração, Transformação e Carregamento, em português), ou *ELT* (*Extract, Load and Transform* - Extração, Carga e Transformação, em português), as ingestões buscam simplificar as chegadas de dados de tamanhos variáveis de forma escalável e buscando a unificação dos dados, possibilitando tratá-los e utilizá-los em análises relevantes para as organizações (Grover, Carey, 2015; Mehan et al, 2017).

3.1.1. ETL

Os processos de ETL surgiram na década de 1970, sendo majoritariamente manuais e suscetíveis a erros, o que demandava grande tempo para sua execução. Com o avanço tecnológico, surgiram ferramentas mais sofisticadas, proporcionando interfaces gráficas para o *design* dos seus fluxos e mecanismos automatizados de controle de erros e otimização. Além disso, a introdução de automações para *design* de *pipelines*, e melhor gerenciamento e monitoramento de erros, tempo e otimização, permitiram uma melhor governança dos dados e maior eficiência nos processos de integração (AWS, 2025; Seenivasan, 2022).

O processo de ETL é uma abordagem tradicional para a integração de dados, sendo amplamente utilizado para consolidar informações de diferentes fontes, como bancos de dados, APIs (*Application Programming Interface* - Interface de Programação de Aplicação, em portugês) e arquivos diversos. Conforme apontado na figura 1, primeiramente os dados são extraídos das fontes de origem, sendo posteriormente

transformados por meio de procedimentos como limpeza, normalização e agregação, garantindo maior qualidade e consistência. Por fim, os dados são salvos em um repositório unificado, possibilitando sua utilização em análises estratégicas (AWS, 2025; Seenivasan, 2022).

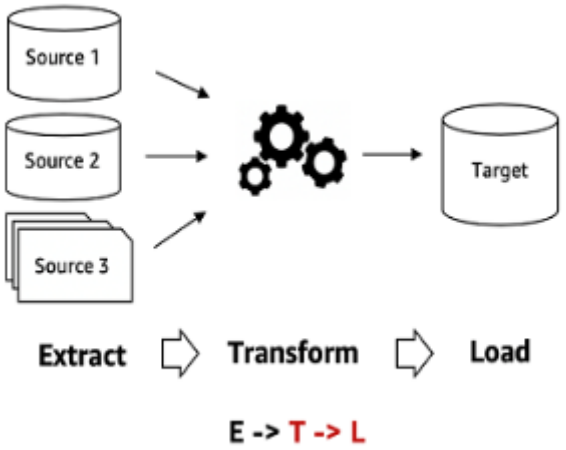


Figura 1 - Representação de um Processo de ETL

Fonte: AWS (2025)

3.1.2. ELT

O processo de ELT é uma variação do modelo do ETL, onde os dados são primeiramente carregados em uma área de preparação dentro do repositório centralizado, ainda em seu formato bruto. Ou seja, as transformações são realizadas diretamente no repositório, aproveitando a capacidade computacional da nuvem para processar grandes volumes de dados. Esse modelo é muito útil, tanto para processamento em lotes quanto em arquiteturas de *streaming*, onde eventos capturados por algoritmos são armazenados e posteriormente transformados para análises avançadas (Reis, Housle, 2022).

Diferente do ETL, onde os dados são transformados antes de serem salvos, no ELT a carga ocorre imediatamente após a extração, armazenando os dados em seu formato bruto para que a transformação aconteça dentro do próprio repositório, conforme ilustrado na figura 2. Essa abordagem permite maior eficiência no tratamento de dados variáveis e facilita a adaptação a mudanças de esquema, além de reduzir o tempo necessário para a ingestão. No entanto, a escolha entre ETL e ELT deve considerar fatores como requisitos de governança, desempenho e capacidade de processamento do ambiente utilizado (Reis e Housle, 2022; AWS, 2025).

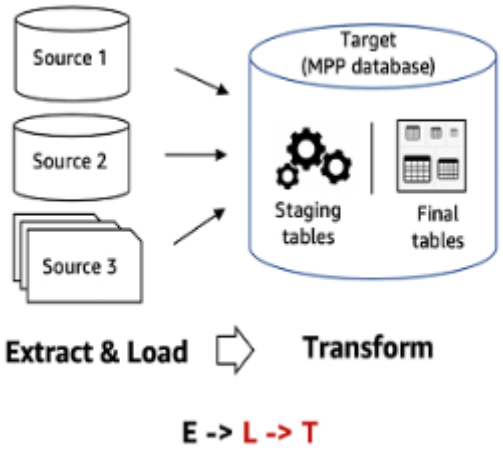


Figura 2. Representação de um Processo de ELT

Fonte: AWS (2025)

3.2. Data Lake

Um *data lake* é um repositório unificado onde as empresas podem ingerir, armazenar, explorar, processar e analisar qualquer tipo ou volume de dados brutos provenientes de fontes diversas, como sistemas operacionais, fontes da *web*, mídias sociais e Internet das Coisas (Tekiner et al, 2021).

De acordo com Reis e Housle (2022), "o *data lake* prometia ser uma força democratizadora, liberando os negócios para beber de uma fonte de dados ilimitados". No entanto, essa arquitetura apresenta desafios significativos, como falta de governança, gerenciamento ineficiente e dificuldades para processar dados de maneira estruturada (Fehad, 2019).

3.3. Data Warehouse

O *data warehouse* é uma opção para organizações que necessitam de dados bem estruturados e governança robusta. Diferente do data lake, o data warehouse é otimizado para consultas analíticas, garantindo alto desempenho e suporte a transações complexas. Ele armazena dados já tratados e organizados, permitindo análises eficientes, porém com menor flexibilidade para lidar com dados brutos e não estruturados (Reis e Housle, 2022).

3.4. Data Lake House

A convergência entre as arquiteturas de data lake e data warehouse levou ao surgimento do *data lake house*, que combina a flexibilidade do data lake com a estruturação e governança do data warehouse. Essa abordagem proporciona um ambiente mais equilibrado, permitindo tanto a ingestão de grandes volumes de dados brutos quanto a realização de análises com performance e governança adequadas, se tornando ideal para este projeto (Reis e Housle, 2022).

3.3. Governança e Segurança em Ambientes de Dados

Armazenar dados provindos de ingestões em sistemas na nuvem oferece vantagens em custo e velocidade de processamento. Porém, a segurança dos dados continua sendo um desafio, que força as empresas a construir suas próprias infraestruturas de armazenamento locais para garantir maior privacidade dos seus dados. Apesar de oferecer mais segurança, essas infraestruturas são caras, exigem uma manutenção qualificada e se tornam mais complexas à medida que a empresa cresce (Tole, 2013).

A segurança de dados se torna um problema complicado na engenharia de dados, principalmente no que se diz a privacidade. As soluções afetam diretamente o sistema e causam dificuldades ao *software*. Portanto, ter um sistema em que há a liberdade do usuário decidir quais informações serão fornecidas, causa problemas aos sistemas de dados (Tole, 2013).

3.4. Arquitetura Medalhão

A Arquitetura Medalhão é um padrão de *design* de dados que organiza as informações em camadas lógicas para melhorar a qualidade e estrutura dos dados ao longo do processo de transformação. Essa arquitetura é tradicionalmente dividida em três camadas: bronze, prata e ouro (figura 3).

A camada bronze armazena os dados brutos extraídos das fontes originais, a camada prata contém dados limpos e transformados e a camada ouro é otimizada para casos de uso específicos e análise avançada. Esse modelo facilita o controle de acesso aos dados e permite a reprocessamento a partir da camada bruta, garantindo maior confiabilidade e flexibilidade na gestão da informação (Wiselka, 2024; Palmer, 2024).

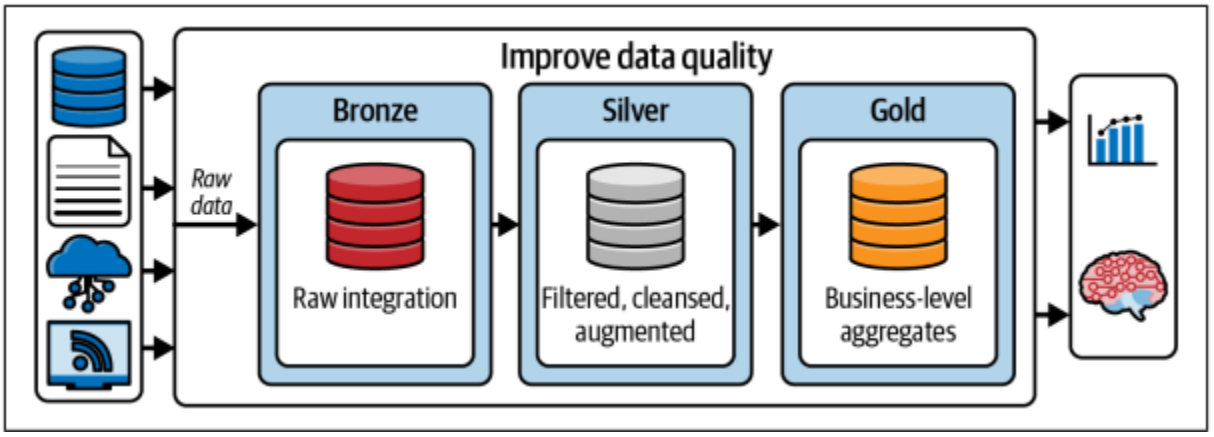


Figura 3. Representação de uma Arquitetura Medalhão

Fonte: Palmer (2024)

3.5. Databricks

O Databricks é uma plataforma que permite que os desenvolvedores desenvolvam projetos e soluções de dados de ponta a ponta, ou seja, viabiliza a ingestão, transformação, processamento, governança e orquestração de dados em uma única plataforma. Como uma solução baseada em um *data lake house*, o Databricks possibilita replicar as funções de um *data warehouse* e um *data lake*, além de adicionar uma camada de governança a partir do Unity Catalog, explicado na seção 3.5.2, e processar grandes quantidades de dados com o Apache Spark, explicado na seção 3.5.1, fornecendo uma plataforma completa de dados (Databricks, 2023; Palmer, 2024).

3.5.1. Apache Spark

A plataforma do Databricks foi criada a partir do *framework* Apache Spark, que viabilizou a utilização da linguagem Python para construção de estruturas personalizadas de ingestão de dados (Databricks, 2023). Seguindo a definição da Apache (2025), o Apache Spark é um "mecanismo que suporta diversas linguagens para executar processos de engenharia de dados, ciência de dados e aprendizado de máquina em máquinas de nó único ou *clusters*".

A partir da possibilidade de utilizar *clusters*, o Spark soluciona o problema de escalabilidade a partir de uma arquitetura distribuída e processamentos em memória, reduzindo drasticamente os tempos de processamento, trazendo a possibilidade de fazer um processamento de dados em paralelo, dividindo o dado em partições e direcionando para cada nó do *cluster* resolver. O *cluster* para este *framework* consiste em um *driver*

que é responsável por coordenar as tarefas e interagir com o usuário final e vários *workers*, que são responsáveis por executar as tarefas recebidas do *driver* (figura 4; Apache, 2025; Wiselka, 2024; Lins et al 2016).

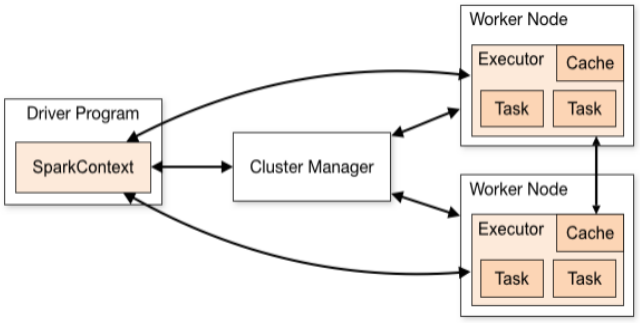


Figura 4. Representação de *Cluster*

Fonte: Apache (2025)

3.5.2. Unity Catalog

O Unity Catalog é uma solução de governança para dados e IA (Inteligência Artificial) no Databricks. De acordo com a documentação oficial da Microsoft (2025), ele "fornece recursos centralizados de controle de acesso, auditoria, linhagem e descoberta de dados em workspaces do Azure Databricks".

Com essa ferramenta, as organizações podem gerenciar dados estruturados e não estruturados em diversos formatos, além de modelos de *machine learning*, *notebooks*, *dashboards* e arquivos em diferentes ambientes de nuvem e plataformas. Essa abordagem permite que cientistas de dados, analistas e engenheiros colaborem de forma segura, garantindo acesso confiável aos dados e IA (Databricks, 2025).

O Unity tem uma hierarquia de 3 níveis observada na figura 5, podendo gerenciar o acesso em cada um deles, desconsiderando o *metastore*, que é um repositório responsável por registrar metadados: O catálogo, sendo o primeiro e mais alto, permite organizar os dados e isolá-los de acordo com um escopo. O esquema ou banco de dados, segundo nível, guarda as tabelas ou outros ativos, ou seja, representa um único caso de uso. Por fim, no nível 3 e mais baixo, têm os volumes lógicos para guardar dados não estruturados e não tabulares, as tabelas de dados, as *views*, as funções e os modelos de IA (Microsoft, 2025).

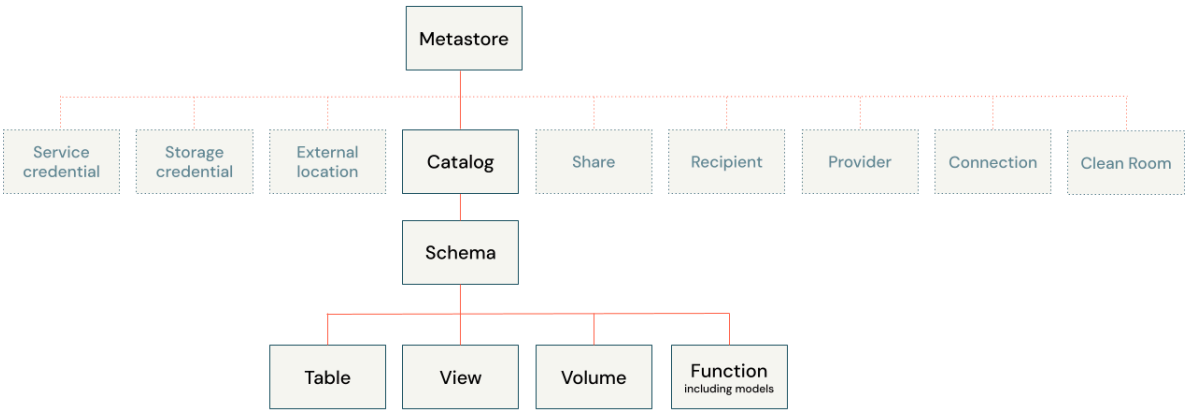


Figura 5. Hierarquia do Unity Catalog

Fonte: MICROSOFT (2025)

3.6. Trabalhos Relacionados

Diversos trabalhos já abordaram soluções para a ingestão de dados, como no trabalho de Damo (2022), que desenvolveram um protótipo de um sistema *web* para ingestão de dados em lotes, centralizando todo o ETL em uma única interface, porém de forma local. Assim como no sistema de Grover, Carey (2015) que utilizam *data feeds,* que são mecanismos para ingestão contínua de dados externos, apresentando arquiteturas para garantir confiabilidade e escalabilidade na ingestão de dados, também de forma local.

Da mesma forma, provedores de serviços em nuvem, como a Microsoft (2024), disponibilizam ferramentas como o Azure Data Factory, que é um serviço de ingestão de dados que permite montar todo o processo de ETL ou ELT apenas com uma interface gráfica. Já a Amazon (2025), disponibiliza o AWS Glue, um serviço de integração de dados que permite processar os dados de diversas fontes e carregá-los em outros destinos, como *data lakes* ou bancos de dados, tanto com interfaces gráficas como por *scripts* em Python com Apache Spark.

Todas as soluções têm o mesmo objetivo, transportar dados de uma ponta a outra. Porém, nas soluções apresentadas por Damo (2023) e Grover & Carey (2015), a escalabilidade é limitada ao poder computacional local. Já nas soluções da Microsoft e Amazon, não há a possibilidade de conectar em fontes que não estejam disponibilizadas para uso comercial, limitando o ELT e forçando as organizações a obterem múltiplos serviços para atender a toda sua necessidade. Isso difere deste trabalho, que segue uma arquitetura completamente cloud e viabiliza adaptar várias fontes de dados caso necessário.

4. Procedimentos Metodológicos

A arquitetura deste trabalho segue o padrão da arquitetura medalhão em um *data lake house* hospedado na Azure, utilizando o Databricks hospedado na região East US 2. O processamento de dados é realizado com Apache Spark e armazenado no formato *delta* em um Azure Data Lake Storage e gerido pelo Unity Catalog, garantindo eficiência e escalabilidade para a ingestão em lotes (figura 6). Além disso, a solução adota o Azure Key Vault, para o gerenciamento seguro de credenciais e segredos necessários para a ingestão (AZURE, 2025).

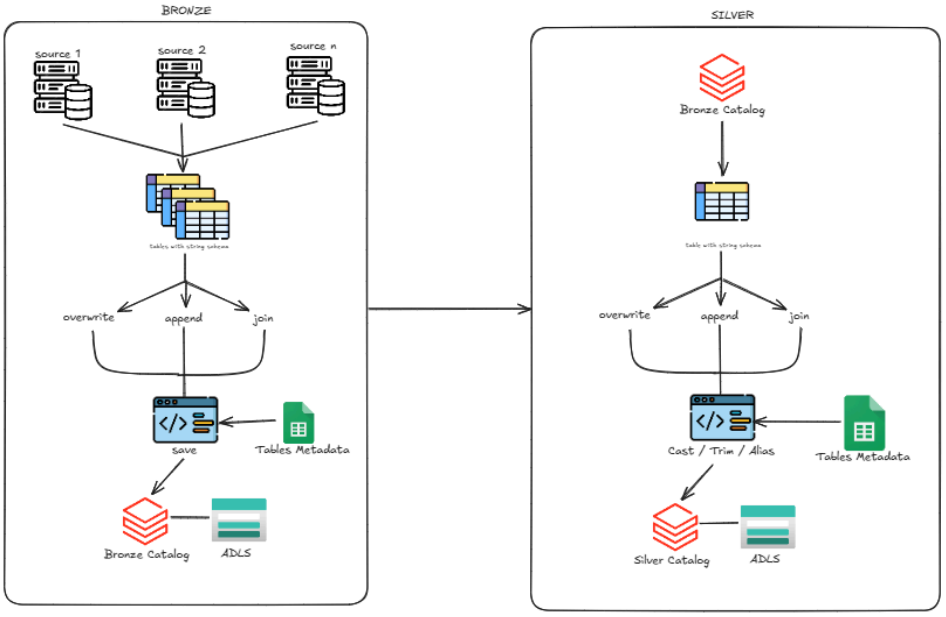


Figura 6. Arquitetura da Ingestão de Dados

Fonte: Elaboração Própria

Para garantir flexibilidade na atualização dos dados, foram implementados três métodos distintos de carregamento das tabelas. O método *overwrite* substitui todos os dados que já existem na tabela pelos novos dados carregados. Por sua vez, o *append* apenas incrementa novos registros à tabela e também atualiza os registros existentes que correspondem aos dados carregados. Por fim, o método *join* realiza a combinação de duas tabelas, uma abordagem útil para efetuar atualizações incrementais baseadas nas relações entre os dados das duas tabelas.

Foram utilizadas três tabelas simuladas em um banco de dados MySQL, sendo uma para cada método, para testar a ingestão de dados. Essas tabelas foram estruturadas para representar um cenário de um ambiente de produção, possibilitando avaliar o

desempenho da ingestão. Os dados fictícios foram gerados com a biblioteca *Faker,* do Python e inseridos no banco MySQL, garantindo variedade e realismo nas simulações (Faker, 2025). Para o método de *overwrite* e *join*, foi gerada uma tabela com 24 colunas do tipo *decimal(12,2)*, 1 coluna do tipo *inteiro*, 8 colunas do tipo *smallint* e 4 colunas do tipo *varchar(255)*, com um total de 64.483.952 linhas. Já no método de *append*, a tabela gerada possui 3 colunas do tipo *date*, 15 colunas *decimal(12,2)*, 5 colunas *inteiro*, 11 colunas *smallint* e 9 colunas *varchar(255)*, com 28.333.521 instâncias (figura 7).

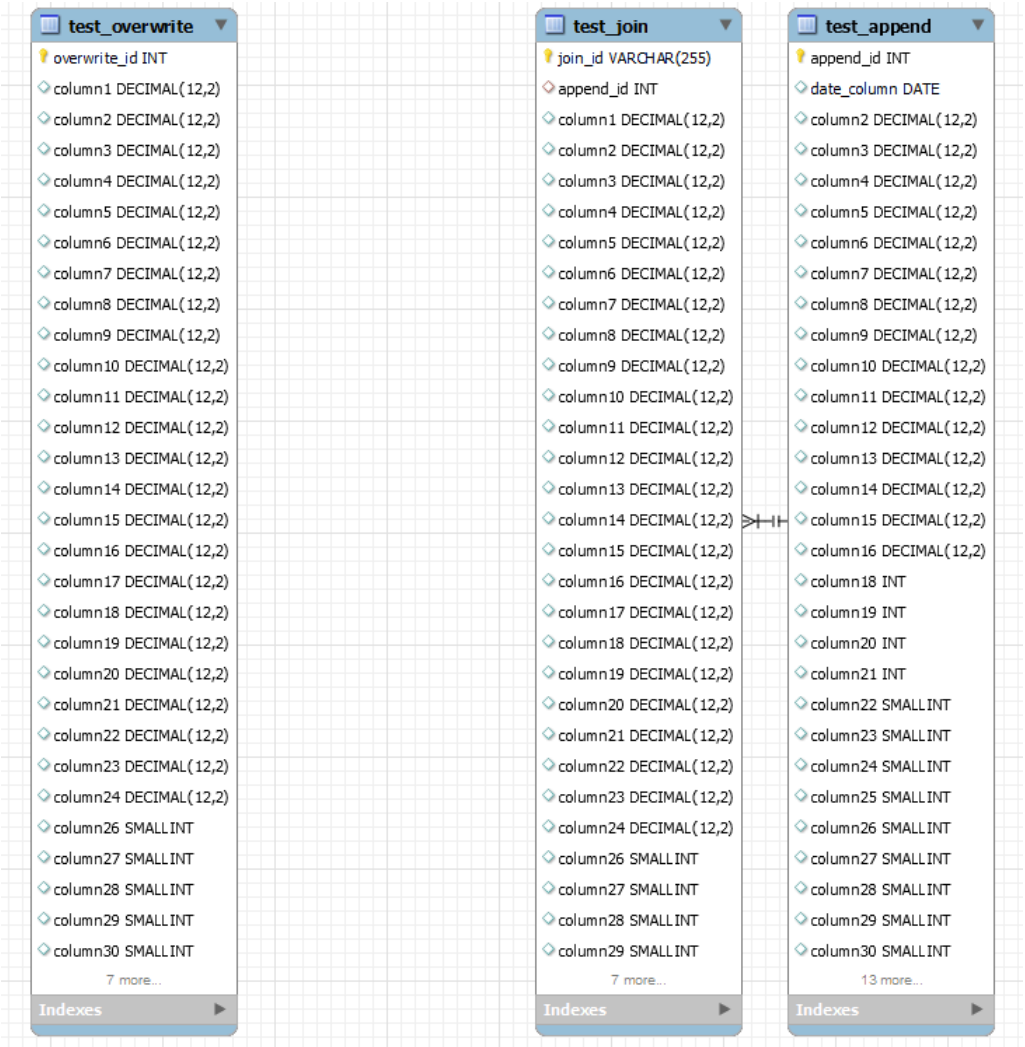


Figura 7. Diagrama Tabelas MySQL

Fonte: Elaboração Própria

Com os testes, foram analisados aspectos como tempo de processamento e possíveis falhas na extração, garantindo que a solução desenvolvida fosse robusta e eficiente para futuras implementações em larga escala.

4.1. Metadados

Para Takahashi (2000), os metadados "são dados que descrevem outros dados". Neste trabalho, os metadados são definidos em duas planilhas do Google Sheets, uma contendo as informações sobre as tabelas e outra contendo as informações das colunas das tabelas (figuras 8 e 9).

source	catalog	schema	table	description	format	date_column	rollback_days
mysql_test	mysql	test	test_overwrite	table for ingestion test	overwrite		
mysql_test	mysql	test	test_append	table for ingestion test	append	date_column	15
mysql_test	mysql	test	test_join	table for ingestion test	join		

id	join_table	join_condition	filters	rls_function	rls_column
overwrite_id				rls	rls_column
append_id				rls	rls_column
join_id	test_append	test_append.append_id == test_join.append_id		rls	rls_column

Figura 8. Planilha das Tabelas

Fonte: Elaboração Própria

table	column	alias	description	data_type
test_overwrite	overwrite_id	OverwriteID	Table ID	int
test_overwrite	column1	Column1	Column 1 for test	string
test_overwrite	column2	Column2	Column 2 for test	string
test_overwrite	column3	Column3	Column 3 for test	string
test_overwrite	column4	Column4	Column 4 for test	string
test_overwrite	column5	Column5	Column 5 for test	string

Figura 9. Planilha das Colunas

Fonte: Elaboração Própria

Estes metadados são acessados com o auxílio da API do Google Sheets com Python e salvos em um catálogo à parte no Unity Catalog dentro do Databricks (figura 10). Por fim, são consumidos dentro da ingestão a partir de um pacote que retorna um dicionário que contém todas as informações das tabelas e utilizados para popular os parâmetros das funções da ingestão.

```python
class GoogleSheetsReader:

    def __init__(self, service_account_credentials: dict):
        self.service_sheets = build(
            "sheets", "v4", credentials=self.__auth(service_account_credentials)
        ).spreadsheets()

    def __auth(self, service_account_credentials: dict):

        creds = Credentials.from_service_account_info(
            service_account_credentials, scopes=
["https://www.googleapis.com/auth/spreadsheets",
"https://www.googleapis.com/auth/drive"]
        )
        return creds

    def read_sheets_to_json(
        self, spreadsheet_id: str, worksheet_name: str, range: str = ""
    ) -> str:
        results = self.service_sheets.values().get(
            spreadsheetId=spreadsheet_id, range=worksheet_name + range
        ).execute()

        headers = results["values"][0]
        values = list(filter(None, results["values"][1:]))

        data = [
            {
                k: v[i].strip() if (i < len(v) and v[i].strip() != "") else None
                for i, k in enumerate(headers)
            }
            for v in values
        ]

        json_data = json.dumps(data)

        return json_data
```

Figura 10. Leitor de Tabelas Google Sheets

Fonte: Elaboração Própria

4.2. Bronze

A camada bronze é a primeira etapa da ingestão, armazenando os dados originais. A leitura é feita a partir do JDBC (*Java Database Connectivity*) para atingir a maior parte dos bancos de dados do mercado atualmente. Todas as colunas no momento da leitura no banco de dados são tratadas como *string*, garantindo que nenhuma informação seja perdida no momento da ingestão (figura 11).

Figura 11. Função de Leitura de Dados

Fonte: Elaboração Própria

O que diferencia os métodos são as consultas SQL utilizadas. No método *overwrite*, todos os dados da tabela são substituídos por novos, o que significa que ele busca todos os registros da tabela no banco de dados (figura 12). Além disso, este método ativa o *Change Data Feed*, que permite rastrear as alterações a nível de linha nas tabelas (DELTA LAKE, 2025). Salienta-se que, caso a tabela ainda não exista, este método é automaticamente chamado para realizar a primeira carga completa da tabela, independente da definição dos metadados.

Figura 12. Funções Método *Overwrite*

Fonte: Elaboração Própria

No método *append*, são buscados apenas os dados mais recentes a partir do dia atual, menos os dias de *rollback* na coluna de data de criação do registro definida nos metadados. Com isso, os dados são salvos na camada bronze sem substituir nenhum registro para obter um rastreio dos lotes que estão sendo inseridos (figura 13).

```
def get_query(table:str, columns:list, filters:str, update_column:str, rollback_days:int,
date_format:str) -> str:
  date_update = (
    datetime.date.today() -datetime.timedelta(rollback_days)
  ).strftime(date_format)
  columns_query = ', '.join(columns)

  query = f"""
    SELECT {columns_query}
      FROM {table}
    WHERE
      {update_column} >= to_date('{date_update}', '{date_format}')
   """
  if filters:
    query += f" AND {filters}"

  return f"({query}) as dbtable"


def write_to_bronze(df:DataFrame, catalog_table:str, update_column:str, rollback_days:str,
date_format:str):
  date_update = (
    datetime.date.today() - datetime.timedelta(rollback_days)
  ).strftime(date_format)

  (
    df
      .write
      .format("delta")
      .mode("overwrite")
      .option("replaceWhere", f"to_date({update_column}) >= to_date('{date_update}')")
      .saveAsTable(catalog_table)
  )
```

Figura 13. Funções Método *Append*

Fonte: Elaboração Própria

Por fim, no método *join*, a tabela é combinada com outra que utiliza o formato *append*, buscando apenas os registros novos a partir da data de criação da tabela de referência. Nesse método, é realizado um *merge*, atualizando os registros existentes e inserindo novos (figura 14).

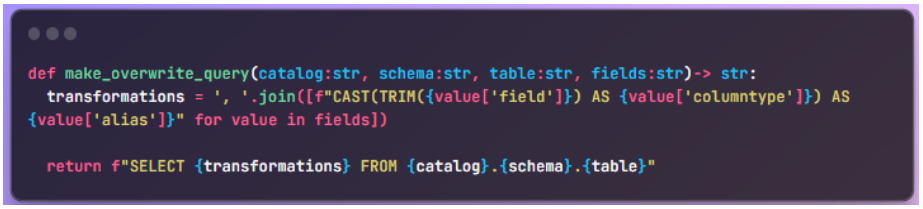
```
def get_query(table:str, join_table:str, join_condition:str, filters:str, columns:list,
update_column:str, rollback_days:int, date_format:str)-> str:
    date_update = (datetime.date.today() -
datetime.timedelta(rollback_days)).strftime(date_format)
    columns_query = ','.join([f'{table}.{i}' for i in columns])
    query = f"""
        SELECT {columns_query}
            FROM {table}
            INNER JOIN {join_table}
                ON {join_condition}
        WHERE {update_column} >= to_date('{date_update}', '{date_format}')
    """

    if filters:
        query += f" AND {filters}"

    return f"({query}"

def write_to_bronze(df, catalog_table:str, keys:list):
  bronze_delta = DeltaTable.forName(spark, catalog_table)
  join_condition = ' AND '.join([f'source.{i} = target.{i}' for i in keys])

  result = bronze_delta.alias('target').merge(
      df.alias('source'),
      join_condition
  ).whenMatchedUpdateAll().whenNotMatchedInsertAll().execute()

  return result
```

Figura 14. Funções Método *Join*

Fonte: Elaboração Própria

4.3. Prata

Após trazer as informações necessárias na camada bronze, é necessário realizar a transformação dos tipos de dados, renomear as colunas para uma nomenclatura legível e remover eventuais espaços excedentes dos valores. Assim como na camada bronze, na camada prata também são utilizados os mesmos métodos para a atualização dos dados que se diferem no SQL realizado e na forma de salvar as informações obtidas. Para o método *overwrite* é feito uma busca da tabela armazenada na camada bronze completa, alterando os tipos, o nome das colunas e realizando um *"trim"*, que remove espaços excedentes no início e no fim das colunas (figura 15).



```
def make_overwrite_query(catalog:str, schema:str, table:str, fields:str)-> str:
  transformations = ', '.join([f"CAST(TRIM({value['field']}) AS {value['columntype']}) AS
{value['alias']}" for value in fields])

  return f"SELECT {transformations} FROM {catalog}.{schema}.{table}"
```

Figura 15. Função SQL Prata Método *Overwrite*

Fonte: Elaboração Própria

Ainda no método de *overwrite*, caso seja a primeira vez que a tabela é ingerida, são aplicadas as descrições, tanto nas colunas quanto nas tabelas, para documentação e entendimento dos dados. Além da aplicação das descrições, é aplicada uma função de RLS (*Row Level Security*), que, segundo Databricks (2025), permite restringir o acesso a linhas de uma tabela com base em atributos do usuário. O RLS criado para esta ingestão de dados faz com que os usuários do grupo *developers*, por exemplo, só visualizem linhas com valor '1' na coluna definida nos metadados, ou seja, *'rls_column'*, enquanto os usuários do grupo *admins* conseguem consultar todos os dados da tabela (figura 16).

```
CREATE OR REPLACE FUNCTION governance.default.rls_function(rls_column STRING)
  RETURN CASE
        WHEN is_account_group_member('developer') THEN (rls_column = '1')
        WHEN is_account_group_member('admins') THEN True
    ELSE False END;
```

Figura 16. Função de RLS

Fonte: Elaboração Própria

Já no método *append*, os novos registros são buscados a partir da versão da tabela *delta*, que é um sistema de gerenciamento de arquivos baseado no formato de arquivos *parquet* (Microsoft, 2025). Com isso, são buscadas apenas as inserções provindas da camada bronze na tabela *delta*, substituindo todas as informações a partir da faixa definida para a data de criação (figura 17).

```
def make_append_query(catalog:str, schema:str, table:str, fields:str, version:int)
-> str:
  transformations = ', '.join([f"CAST(TRIM({value['field']}) AS
{value['columntype']}) AS {value['alias']}" for value in fields])

  return f'SELECT {transformations} FROM table_changes("{catalog}.{schema}.
{table}", {version}) WHERE _change_type = "insert"'
```

Figura 17. Função SQL Prata Método *Append*

Fonte: Elaboração Própria

Por fim, o método *join* busca a tabela inteira da camada bronze e realiza um *merge* com base no ID definido nos metadados na tabela na camada prata. Com isso, os dados novos são inseridos e os antigos são atualizados (figura 18).

```
def make_join_query(catalog:str, schema:str, table:str, fields:list)-> str:
    transformations = ', '.join([f"CAST(TRIM({value['field']}) AS
{value['columntype']}) AS {value['alias']}" for value in fields])

    return f"SELECT {transformations} FROM {catalog}.{schema}.{table}"

def silver_join_pipeline(bronze_catalog:str, silver_catalog:str, schema:str,
table:str, keys:list, fields:list):

  query = make_join_query(bronze_catalog, schema, table, fields)
  df = spark.sql(query)

  silver_delta = DeltaTable.forName(spark, f"{silver_catalog}.{schema}.{table}")
  silver_keys = [field['alias'] for field in fields if field['field'] in keys]
  join_condition = ' AND '.join([f'source.{i} = target.{i}' for i in silver_keys])

  result = silver_delta.alias('target').merge(
      df.alias('source'),
      join_condition
  ).whenMatchedUpdateAll().whenNotMatchedInsertAll().execute()

  return {'query':query}
```

Figura 18. Funções Prata Método *Join*

Fonte: Elaboração Própria

4.4. Monitoramento

O monitoramento das tabelas acontece por meio de *logs* de ingestão de cada tabela. Esses *logs* são enviados para o Datadog, que é uma ferramenta que permite unificá-los e métricas para viabilizar a rastreabilidade da ingestão (Datadog, 2025).

São enviados por meio da integração do Datadog com a linguagem Python com as informações descritas no Quadro 1.

Quadro 1. Esquema de *logs*

Indicador	Descrição
Serviço	O nome do serviço que gerou o *log*.
Método	O método de ingestão utilizado para a tabela.
Bronze Query	A query que foi realizada para consulta no banco de dados e popular a camada bronze.
Silver Query	A query que foi realizada para consulta na camada bronze e popular a camada prata.
Data e hora de início	Data e hora de início da ingestão da tabela.
Data e hora de fim	Data e hora de fim da ingestão da tabela.
Status	Indica se o *log* é um erro, aviso ou informação.
Nome da tabela	Informa o nome da tabela que passou pela ingestão.

5. Implementação

Foram utilizadas as 3 tabelas simuladas, uma para cada método de atualização da tabela, que demonstram eficácia nos diferentes métodos de carga: *overwrite*, *append* e *join*. A computação utilizada foi uma E8ds na versão 5 com 2 *workers* e o *driver*, que possuem 64 gigabytes de memória cada um. O banco de dados foi hospedado em uma máquina equipada com processador Intel Core i5-8100, 32 GB de memória RAM e 500GB de armazenamento em SSD. A escolha do método influenciou diretamente o tempo de processamento dos dados e quantidade de linhas extraídas (figura 19 e 20).

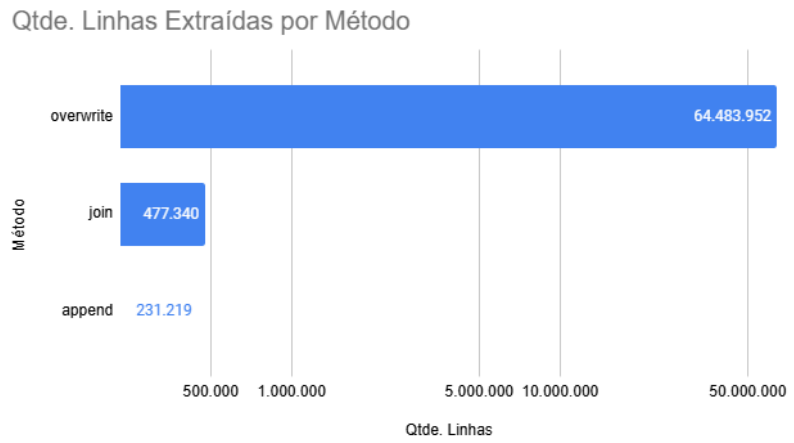


Figura 19. Quantidade de Linhas Extraídas por Método

Fonte: Elaboração Própria

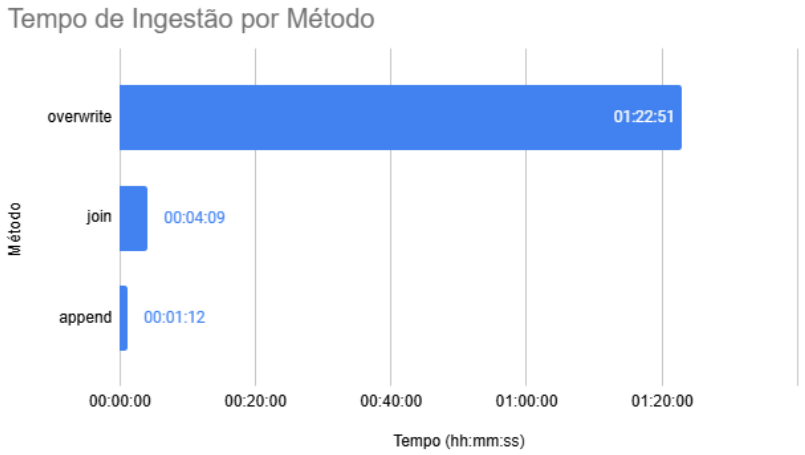


Figura 20. Tempo de Extração por Método

Fonte: Elaboração Própria

O método de *overwrite* permitiu substituir completamente todos os dados antigos por novos, garantindo a integridade e a atualização da base. No entanto, o tempo de extração para esta tabela foi de 1 hora, 22 minutos e 51 segundos, sendo significativamente alto, exigindo mais recursos computacionais. Este método se mostra recomendado para cenários onde a tabela precisa ser recriada regularmente ou para tabelas com poucas quantidades de linhas e colunas, onde fica mais simples e barato computacionalmente buscar todos os registros da tabela.

O principal resultado desse método se dá no tempo de carga e quantidade de registros buscados, atualizando os registros da tabela em apenas 1 minuto e 13 segundos e buscando uma quantidade de linhas muito menor que no método de *overwrite*, já que

buscou somente dados criados nos últimos 15 dias. Portanto, o método se mostra eficiente para cargas incrementais permitindo a inserção de novos registros e atualização das informações que sofreram alterações dentro da faixa de dias definida.

A utilização do método *join* permite consolidar dados de tabelas diferentes para otimizar o processo de atualização de registros. O tempo de processamento foi de 4 minutos e 9 segundos, sendo menor que o método *overwrite*, mas maior que o método *append*, evidenciando o maior esforço computacional para combinar os registros. Porém, esse método se apresenta essencial para integração dos dados e otimização para tabelas que não possuem algum campo de data de criação para utilizar o método *append*.

Portanto, observa-se que, a ingestão consegue fazer a atualização dos dados de forma íntegra e possibilita consolidar informações de diferentes bancos de dados em um tempo considerável, contanto que haja a possibilidade de conectá-lo via JDBC. Entretanto, a necessidade de muitos recursos computacionais ainda é um problema, visto que uma grande quantidade de dados são obtidos mesmo que não sejam necessários.

6. Conclusão

Este trabalho abordou a ingestão de dados em uma plataforma cloud para possibilitar a análise de dados, evidenciando a relevância do Databricks como uma ferramenta robusta para ingestão de dados em larga escala, permitindo otimização conforme a necessidade de cada ambiente. A escolha entre os métodos *overwrite*, *append* e *join* deve ser guiada a partir de requisitos específicos de cada fonte, equilibrando performance, integridade e custo computacional.

O método *overwrite* mostrou-se adequado para cenários onde a atualização completa é necessária apesar do seu alto tempo de execução ou em tabelas muito pequenas. O *append* se destacou como uma opção eficiente para cargas incrementais, podendo ter uma retenção de alguns dias definidos pelo usuário, melhorando a performance pelo fato de buscar apenas dados novos. Por fim, o método *join* permitiu consolidar diferentes tabelas para obter um ganho de performance e trazer apenas dados novos da tabela em questão.

Dessa forma, a utilização da ingestão de dados proposta dentro de um *pipeline* no Databricks possibilitou a construção de um ambiente de dados escalável, com um bom desempenho e íntegro, contribuindo para análises de dados eficientes e assertivas

no contexto empresarial e acadêmico. Como trabalhos futuros, recomenda-se a investigação de técnicas mais avançadas de otimização, como o *liquid clustering* para otimizar as leituras da camada bronze para a prata e o *adaptive query execution* para aprimorar a eficiência das cargas de dados, além da adição de métodos de CDC (*Change Data Capture*) para viabilizar análises de dados em tempo real.

7. Referências

AMAZON. **AWS Glue**. Disponível em: <https://aws.amazon.com/pt/glue/>. Acesso em: 18 mar. 2025.

APACHE. **What is Apache Spark™?.** Disponível em: <https://spark.apache.org/>. Acesso em: 18 mar. 2025.

AWS. ETL x ELT — **Diferença entre abordagens de processamento de dados**. Disponível em: <https://aws.amazon.com/pt/compare/the-difference-between-etl-and-elt/>. Acesso em: 18 mar. 2025.

AZURE. **Azure Key Vault.** Disponível em: <https://azure.microsoft.com/pt-br/products/key-vault>. Acesso em: 08 jun. 2025.

Cândido, Ana Clara; Araújo Junior, Rogério Henrique de. Potencialidades do desenvolvimento de cloud computing no âmbito da gestão da informação. **Perspectivas em Ciência da Informação**, v. 27, 2022. DOI: <https://doi.org/10.1590/1981-5344/25731>. Acesso em: 18 mar. 2025.

Damo et al, Juan Vinícius Casagrande. **Sistema de Ingestão de dados para datalake**. 2022. Disponível em: <https://ti.faccat.br/wp-content/uploads/2023/02/Juan-Vinicius-Casagrande-Damo.pdf>. Acesso em: 12 mar. 2025.

DATABRICKS. **Filter sensitive table data using row filters and column masks**. Disponível em: <https://docs.databricks.com/aws/en/tables/row-and-column-filters>. Acesso em: 22 abr. 2025.

DATABRICKS. **O livro completo da engenharia de dados**. 2. ed. 2023. Disponível em: <https://www.databricks.com/br/resources/ebook/big-book-data-engineering-2nd-edition>. Acesso em: 18 mar. 2025.

DATABRICKS. **Produto**. Disponível em: <https://www.databricks.com/br/product/data-intelligence-platform>. Acesso em: 22 abr. 2025.

DATABRICKS. **Unity Catalog**. Disponível em: <https://www.databricks.com/br/product/unity-catalog>. Acesso em: 22 abr. 2025.

DATADOG. **Datadog Documentation**. Disponível em: <https://docs.datadoghq.com/>. Acesso em: 12 mar. 2025.

DELTA LAKE. **Change data feed**. Disponível em: <https://docs.delta.io/latest/delta-change-data-feed.html>. Acesso em: 22 abr. 2025.

FAKER. **Faker documentation**. Disponível em: <https://faker.readthedocs.io/en/master/>. Acesso em: 23 abr. 2025.

Fehad, Pervaiz et al. **Examining the challenges in development data pipeline**. 2019. p. 13-21. DOI: <https://doi.org/10.1145/3314344.3332496>. Acesso em: 2 abr. 2025.

Grover, Raman; Carey, Michael J. **Data Ingestion in AsterixDB**. EDBT. 2015. p. 605-616.

IBM. **Pipeline de dados**. 2024. Disponível em: <https://www.ibm.com/br-pt/topics/data-pipeline>. Acesso em: 20 fev. 2025.

Lin, Erica et al. **Global optimization of data pipelines in heterogeneous cloud environments**. 2022. DOI: <https://doi.org/10.48550/arXiv.2202.05711>. Acesso em: 20 mar. 2025.

Lins, Maurício et al. **Estudo de caso de processamento de ETL em plataforma Big Data**. 2016. Disponível em: <https://www.researchgate.net/publication/340296179_ESTUDO_DE_CASO_DE_PROCESSAMENTO_DE_ETL_EM_PLATAFORMA_BIG_DATA>. Acesso em: 20 mar. 2025.

Meehan, John et al. **Data Ingestion for the Connected World**. In: Cidr. 2017. p. 8-11.

MICROSOFT . **O processo de ingestão com a análise de escala de nuvem no Azure - Cloud Adoption Framework**. Disponível em: <https://learn.microsoft.com/pt-br/azure/cloud-adoption-framework/scenarios/cloud-scale-analytics/best-practices/data-ingestion>. Acesso em: 20 fev. 2025.

MICROSOFT. **O que é Catálogo do Unity?**. Disponível em: <https://learn.microsoft.com/pt-br/azure/databricks/data-governance/unity-catalog>. Acesso em: 22 abr. 2025.

Palmer, Matt. **Understanding ETL Data Pipelines for Modern Data Architectures**. 1. ed. O'Reilly Media, 2024.

Pedrosa, Paulo HC; Nogueira, Tiago. **Computação em nuvem.** 2011. Disponível em: <https://ic.unicamp.br/~ducatte/mo401/1s2011/T2/Artigos/G04-095352-120531-t2.pdf>. Acesso em: 22 abr. 2025.

Rapôso, Cláudio Filipe Lima et al. Impactos da Computação em Nuvem na Arquitetura de Software: Uma Análise de Literatura. **Revista Tópicos**, v. 2, n. 14, 2024, p.

1-12. DOI: <https://doi.org/10.5281/zenodo.13949416>. Acesso em: Acesso em: 20 fev. 2025.

Reis, Joe; Housley, Matt. **Fundamentos de engenharia de dados**. In: São Paulo: Novatec. 2023.

Sagiroglu, Sarej; Sinac, Duygu **Big data: a review**. In: International Conference on Collaboration Technologies and Systems (CTS). 2013. p. 42–47. DOI: <https://doi.org/10.1109/CTS.2013.6567202>. Acesso em: 21 mar. 2025.

Seenivazan, Dhamotharan. **ETL vs ELT: Choosing the right approach for your data warehouse**. In: International Journal for Research Trends and Innovation, 2022. DOI: <http://dx.doi.org/10.6084/m9.doione.IJRTI2202018>. Acesso em: 8 maio 2025.

Takahashi, Tadao. **Sociedade da informação no Brasil: o livro verde**. Brasília: Ministério da Ciência e Tecnologia, 2000.

Taurion, Cezar. **Big data**. Sp: Brasport, 2013.

Tekiner, Firat; Levy, Rachel; Pierce, Susan. **Converging Architectures: Bringing Data Lakes and Data Warehouses Together**. Google Cloud. 2021.

Tole, Alexandru Adrian. **Big Data Challenges**. Database Systems Journal, v. 4, 2013.

Wiselka, Michal. **Development of Modern Data Platform using Medallion Architecture**. 2024.