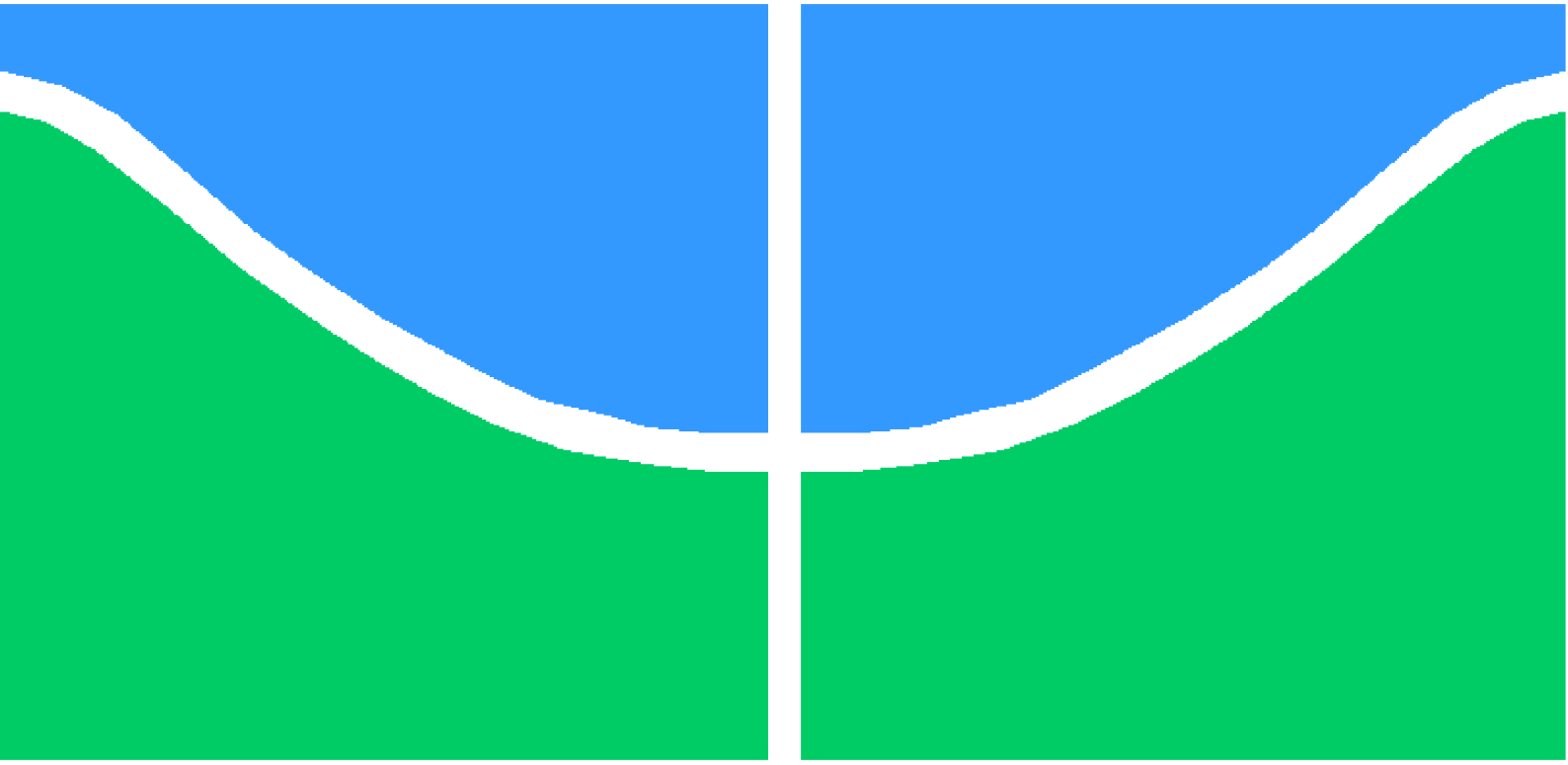


Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Software Engineering

Construindo um processo ETL de Streaming utilizando o Ferramentas Proprietárias do Databricks e comparando-a com uma abordagem de ETL Tradicional

Autor: Thiago Ferreira

Orientador: Dr. Fabrício Ataides Braz

Brasília, DF

2023

Thiago Ferreira

Construindo um processo ETL de Streaming utilizando o Ferramentas Proprietárias do Databricks e comparando-a com uma abordagem de ETL Tradicional

Monografia submetida ao curso de graduação em (Software Engineering) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Software Engineering).

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Dr. Fabrício Ataides Braz

Brasília, DF

2023

Thiago Ferreira

Construindo um processo ETL de Streaming utilizando o Ferramentas Proprietárias do Databricks e comparando-a com uma abordagem de ETL Tradicional

Monografia submetida ao curso de graduação em (Software Engineering) da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em (Software Engineering).

Trabalho aprovado. Brasília, DF, 17 de Fevereiro de 2023:

Dr. Fabrício Ataides Braz
Orientador

Dr. Nilton Correia da Silva
Convidado 1

Dr. Henrique Marra Taira Menegaz
Convidado 2

Brasília, DF
2023

Dedico este trabalho a mim mesmo, pois só eu sei o quão difícil foi para chegar aqui e também aos meus pais, que me apoiaram durante todo o processo, assim como aos amigos que fiz durante a graduação, sem eles eu talvez não tivesse chegado até aqui.

Agradecimentos

Gostaria de agradecer a todos que participaram dessa jornada, de talvez, a coisa mais difícil que eu fiz até o momento em minha vida. Agradeço a quem me ajudou e também a quem me prejudicou, pois aprendi com as derrotas e os erros muito mais do que com as vitórias, me tornei mais forte.

Gostaria de agradecer também ao professor Fabrício por me orientar e aceitar me orientar as várias vezes que eu tive de fazer esse processo e me auxiliar durante todo o percurso para chegar aqui.

"Sofremos mais em imaginação do que em realidade.
(Sêneca)

Resumo

Assim como o dado é tido como o novo petróleo, de nada serve o dado inexplorável. Com o advento da era do Big Data e a crescente demanda por soluções aplicáveis à imensidão de dados gerada pela humanidade hoje, novas ferramentas tendem a surgir. Uma das principais frentes do ambiente de engenharia de dados tem sido tomada por uma empresa chamada Databricks.

A Databricks fornece uma plataforma de engenharia de dados com diversas ferramentas proprietárias para ingestão, processamento e exposição de dados. Neste trabalho focaremos nas ferramentas de Streaming de Dados, o Auto-Loader e as Delta Live Tables do Databricks. Para ter uma base em como essas ferramentas se comparam com abordagens tradicionais, neste trabalho são construídas soluções tanto para as ferramentas do Databricks quanto para uma abordagem tradicional com Spark e Kafka e concluímos se a solução do Databricks é um diferencial tão grande quanto ela se propõe a ser.

Palavras-chaves: databricks, data warehouse, kafka, auto-loader, delta lake, delta table, delta live table, etl, elt, stream, streaming, data lake, data lakehouse, performance, eventsim, spark, spark structured streaming, data engineering, pipeline, delta live tables, stream de dados, engenharia de dados.

Abstract

Just as data is considered the new oil, unexplorable data is useless. With the advent of the Big Data era and the growing demand for solutions to the vast amount of data generated by humanity today, new tools are likely to emerge. One of the main fronts of the data engineering environment has been taken by a company called Databricks.

Databricks provides a data engineering platform with various proprietary tools for data ingestion, processing, and exposure. In this work, we will focus on the Data Streaming tools, the Auto-Loader, and the Databricks Delta Live Tables. To have a basis for comparing these tools with traditional approaches, this work builds solutions for both Databricks tools and a traditional approach with Spark and Kafka, and concludes whether the Databricks solution is as much of a differential as it sets out to be.

Key-words: databricks, data warehouse, kafka, auto-loader, delta lake, delta table, delta live table, etl, elt, stream, streaming, data lake, data lakehouse, performance, etl design, eventsim, spark, spark structured streaming, data engineering, pipeline, delta live tables, data stream.

Lista de ilustrações

Figura 1 – Arquitetura Multi-Hop para Delta Lakes 17
Figura 2 – Performance de uma regressão logística em Hadoop MapReduce vs. Spark para 100GB de dados em 50 máquinas do m2.4xlarge EC2. . . . 20
Figura 3 – Representação gráfica de um processo ETL 21
Figura 4 – Fluxograma do Plano Metodológico 23
Figura 5 – *SQL* para criação da tabela *bronze*, utilizando o *Databricks Auto-Loader* 30
Figura 6 – SQL para criação da tabela bronze, utilizando o *Databricks Auto-Loader* 30
Figura 7 – Código SQL para a tabela de prata 'tweets_en', contendo os tweets captados em inglês. 32
Figura 8 – Código SQL para a tabela de prata 'tweet_pt', contendo os tweets captados em português. 32
Figura 9 – Código SQL para a tabela de prata 'tweets_languages', a contagem para cada linguagem dos tweets captados em streaming. 33
Figura 10 – SQL para a criação da tabela dourada tweets_language_count, sendo comporta por tweets_pt e tweets_languages. 33
Figura 11 – Pipeline de dados representado pela interface das delta live tables . . . 33
Figura 12 – Pipeline de dados representado a dinâmica dos consumers, producers, a stream e o Spark 36

Lista de tabelas

Tabela 1 – Schema da tabela Bronze. 31

Lista de abreviaturas e siglas

ETL	Extract, Transform, Load
ELT	Extract, Load, Transform
DW	Data Warehouse
DLT	Delta Live Table
SQL	Standard Query Language
CSV	Comma Separated Values
JSON	JavaScript Object Notation
SKU	Stock Keeping Unit
API	Application Programming Interface
LGPD	Lei Geral de Proteção de Dados Pessoais
NoSQL	Not Only SQL
RAM	Random Access Memory
AWS	Amazon Web Services
GCP	Google Cloud Provider

Sumário

1 INTRODUÇÃO . . . 13
1.1 Contextualização . . . 13
1.2 Problema . . . 13
1.3 Objetivos Gerais . . . 14
1.3.1 Objetivos Específicos . . . 14
1.4 Estrutura do Documento . . . 14

2 REFERENCIAL TEÓRICO . . . 15
2.1 Databricks . . . 15
2.1.1 Auto-Loader . . . 15
2.1.2 Delta Lakes . . . 16
2.1.3 Arquitetura Multi-Hop . . . 16
2.1.3.1 Delta Table . . . 17
2.1.3.2 Delta Live Tables . . . 17
2.2 Soluções de Armazenamento . . . 18
2.2.1 Data Warehouse . . . 18
2.2.2 Data Lakes . . . 18
2.2.3 Data Lakehouse . . . 18
2.2.4 Apache Kafka . . . 19
2.3 Apache Spark . . . 19
2.4 ETL . . . 21

3 MATERIAIS E MÉTODOS . . . 22
3.1 Considerações Iniciais . . . 22
3.2 Plano Metodológico . . . 22
3.2.1 Definição do processo a ser construído . . . 22
3.2.2 Definição das Ferramentas e Tecnologias a serem utilizadas . . . 23
3.2.3 Escolhendo Uma Fonte de Dados . . . 23
3.2.3.1 Event Sim Stream Simulator . . . 24
3.2.3.2 Twitter API . . . 24
3.2.4 Construção de um fluxo de streaming . . . 24
3.2.5 Construção de um pipeline de refinamento e armazenamento de dados . . . 24
3.2.6 Construção de uma solução Apache Kafka/Spark Structured Streaming . . . 25
3.2.7 Comparação avaliativa dos métodos, abordagens e resultados obtidos . . . 25
3.2.8 Conclusão quanto aos resultados e a pesquisa . . . 26
3.3 Referências . . . 26

3.4 Considerações Finais . . . 26

4 RESULTADOS E DISCUSSÃO . . . 27
4.1 Definição do Processo a Ser Construído . . . 27
4.2 Definição das ferramentas e tecnologias a serem utilizadas . . . 27
4.2.1 Linguagem de Programação . . . 27
4.2.2 Soluções de Armazenamento e Gestão de Dados . . . 27
4.2.3 Databricks . . . 28
4.2.4 Apache Spark . . . 28
4.3 Definição de uma fonte de dados . . . 28
4.3.1 EventSim Stream Simulator e Databricks Autoloader . . . 28
4.3.2 Twitter API . . . 28
4.4 Estruturando o Processo de Streaming . . . 29
4.4.1 Construção de um Pipeline de refinamento e armazenamento de dados . . . 30
4.5 Construção de uma solução Apache Kafka/Spark Structured Streaming . . . 34
4.5.1 O Provisionamento do Ambiente . . . 34
4.5.2 Criação de Tópico . . . 34
4.5.3 Criação do Producer . . . 35
4.5.4 Criação do Consumer . . . 35
4.6 Comparação avaliativa dos métodos, abordagens e resultados obtidos 36
4.6.0.1 Facilidade de Configuração e Manutenção . . . 36
4.6.1 Escalabilidade . . . 37
4.6.2 Tolerância a Falhas . . . 38
4.6.3 Latência . . . 39
4.6.4 Integração com outras ferramentas . . . 39
4.6.5 Segurança e Conformidade . . . 40
4.6.6 Custo . . . 40
4.7 Conclusão quanto aos resultados e a pesquisa . . . 41

REFERÊNCIAS . . . 44

1 Introdução

1.1 Contextualização

Desde a década de 80, *Data Warehousing e Business Intelligence* têm sido os protagonistas na transformação das empresas em empresas *"Data-Driven"*.(KIMBALL; ROSS, 2013) Embora tenham ocupado o posto de líderes há muito tempo, as *Data Warehouses* vêm sendo desafiadas por tecnologias mais novas e flexíveis, como *Data Lakes e Data Lakehouses*, que têm sido uma tendência crescente no mercado recentemente.

A empresa líder nesse crescimento da adoção das *Data Lakehouses* é a *Databricks*, fundada pelo criador do *Spark*, Matei Zahari, e atualmente detém **10.63%** do mercado de Big Data, ou **22.06%** se considerarmos o *Azure Databricks* neste cálculo.(DATABRICKS. . . , a) A *Databricks* oferece uma plataforma integrada para todos os aspectos relacionados a dados, incluindo Engenharia de Dados, *Analytics*, Ciência de Dados e *Machine Learning*, e tem fornecido inovações tecnológicas em muitos processos em cada uma dessas áreas.(DATA. . . , 2023)

Na Engenharia de Dados, o *Databricks* simplifica o processo de ingestão de dados com ferramentas proprietárias, como o *Databricks Auto-Loader* e *Delta Live Tables*, que prometem mudar a forma como a ingestão de dados é realizada e torná-la mais fácil. No entanto, antes da popularização da plataforma, métodos de ingestão de *stream* com *Apache Kafka e Spark Structured Streaming* já eram usados como padrão na indústria de *Big Data* e mesmo empresas que usam o *Databricks* ainda mantêm seus métodos em *Spark e Kafka*.(APACHE. . . ,)

1.2 Problema

Este trabalho visa investigar a eficácia dos novos processos de ingestão de dados de streaming propostos pela *Databricks*, especificamente a utilização de *Delta Live Tables* e *Databricks Auto-Loader*, em comparação com métodos tradicionais. Embora esses novos processos sejam promissores, existe uma escassez de informações a respeito de sua eficiência e efetividade na realização de tarefas de ingestão de dados.

Para responder a esta questão, será realizada uma comparação entre os métodos tradicionais, baseados em *Apache Spark e Apache Kafka*, e os novos processos propostos pela *Databricks*. Desta forma, espera-se obter uma visão geral do processo de ingestão de dados de *streaming*, identificar as principais diferenças entre os métodos e avaliar qual é o método mais eficiente, performático e fácil de ser utilizado. Ao final deste estudo, espera-

se contribuir para a tomada de decisões mais informadas quanto à escolha da abordagem mais adequada para a ingestão de dados de *streaming*.

1.3 Objetivos Gerais

O objetivo deste trabalho consiste em construir um *ETL de streaming* utilizando o *Databricks* e suas tecnologias proprietárias: *Auto-Loader e Delta Live Tables*, assim como construir a mesma solução utilizando os tradicionais métodos com *Apache Kafka e Spark Structured Streaming* e assim, avaliar as diferenças e benefícios entre os métodos.

1.3.1 Objetivos Específicos

- Construir um Pipeline de *Streaming* utilizando *Databricks* Auto-Loader e Delta Live Tables
- Construir um pipeline de *Streaming* usando *Spark Structured Streaming e Apache Kafka*
- Encontrar uma fonte de dados que supra as necessidades do projeto.
- Modelar o Design e Arquitetura do Processo de ETL desenvolvido.
- Avaliar comparativamente as abordagens utilizadas.

1.4 Estrutura do Documento

Este trabalho de conclusão de curso está organizado nos seguintes capítulos:

- **Capítulo 1 - Introdução:** O capítulo 1 apresenta a contextualização do trabalho, sua justificativa, os objetivos gerais e específicos e a estrutura do documento
- **Capítulo 2 - Referencial Teórico:** Neste capítulo são especificados os fundamentos conceituais que possibilitam o trabalho.
- **Capítulo 3 - Materiais e Métodos:** Apresenta de forma mais aprofundada a metodologia adotada no projeto e as etapas executadas durante o decorrer deste projeto, caracterizando também, o objeto de estudo.
- **Capítulo 4 - Resultados e Discussão:** Aqui expomos os resultados obtidos durante a execução deste trabalho e discutimo-os.

2 Referencial Teórico

2.1 Databricks

Databricks é uma plataforma de análise de *Big Data* baseada em nuvem, desenvolvida pela empresa de mesmo nome. É utilizada globalmente por empresas e organizações para processar e analisar grandes quantidades de dados. A plataforma oferece diversas funcionalidades, como Análise de Dados, Ciência, Engenharia e *Machine Learning*, bem como integração direta com o *Spark* e o *framework Delta Lake*.(DATA. . . , 2023)

A plataforma permite o desenvolvimento em núvem, com *clusters* interativos e execução de jobs. Os notebooks integrados do *Databricks*, com suporte para várias linguagens como Python, R, Scala e SQL, são um dos principais destaques da plataforma. Eles são de formato proprietário, mas podem ser exportados e compatíveis com desenvolvimento on-premise.(DATABRICKS. . . , b)

O Databricks é projetado para lidar com grandes volumes de dados, usuários e processamento, com alta capacidade de processamento paralelo e capacidade de suportar muitos processos e usuários simultâneos. A integração de dados também é uma forte característica da plataforma, com a capacidade de ingerir dados de fontes estruturadas ou não e integrar *Data Lakes* e *Data Warehouses* em um ambiente único de *Data Lakehouse*. Além disso, a plataforma possui funcionalidades exclusivas associadas ao *Spark*, como *Delta Live Tables* e *Autoloader*.(DATABRICKS. . . , b)

2.1.1 Auto-Loader

Segundo (L'ESTEVE, 2022) o *Auto-Loader* fornece uma fonte para o Spark Structured Streaming chamado *cloudFiles*. O *cloudFiles* fornece a capacidade de processamento incremental em diversos formatos, tais como *CSV, JSON e PARQUET*, ao mesmo tempo que é gerenciado a evolução de *schema* do dado de *streaming*, armazenando-o em um *DataFrame*.

Sendo assim, um *Auto-Loader* é uma funcionalidade que permite a ingestão automática de dados em um *data lake, warehouse ou lakehouse*. Sendo assim, a ferramenta é capaz de automatizar o processo de ingestão de dados, cortando a necessidade de intervenção humana na execução.

2.1.2 Delta Lakes

Delta Lake é um projeto *open source* que possibilita a construção de *Lakehouses* em cima de *Data Lakes*. O *Delta Lake* possibilita o uso de transações **ACID**, gerenciamento de metadados, unifica soluções de *streaming* e *batch* em cima de *data lakes* tais como *S3*. Também é possibilitado por *delta lakes*, o versionamento de dados, podendo dar *rollback* e mantendo a auditoria dos processos. Toda essa arquitetura é aplicada no *Databricks*, trazendo uma solução mais eficaz, a prova de falhas e sólida para o ecossistema da plataforma.(MSSAPERLA,)

2.1.3 Arquitetura Multi-Hop

A *Arquitetura Multi-Hop* é baseada em um sistema de tabelas, onde são utilizadas tabelas classificadas como *bronze, silver e gold* para representar a qualidade dos dados da tabela e o grau de refinamento pelo qual aquele dado passou. O sistema *multi-hop* é definido por um tabela verdade da qual tudo flui.(HEINTZ; LEE,) Essa tabela servirá de fonte para alimentar a tabela *Silver* que possuirão um grau maior de refinamento, passando por alguns tratamentos e sendo mais seletivas quanto aos dados representados, sendo estes um subconjunto dos dados da tabela verdade, a bronze.

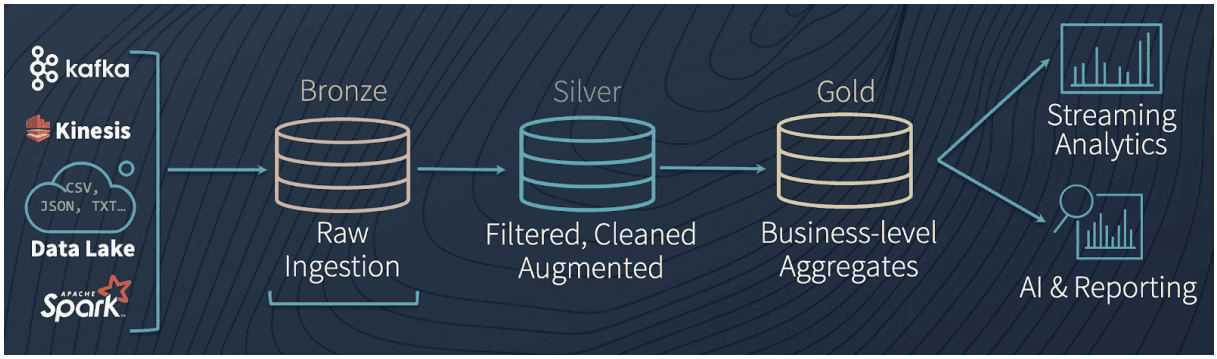
A tabela de Bronze é uma representação crua do dado, sendo este representado da forma como estava na fonte, sem refinamento algum. Esta tabela pode ser criada através de diversas fontes de dados e ferramentas, dentre estas está a ingestão de arquivos **CSV**, **JSON** ou **TXT**, uma fonte Kafka, uma *stream Kinesis* ou através do *Spark Structured Streaming*.

A tabela de Prata normalmente representa um subconjunto de uma tabela de qualidade bronze, selecionando dados específicos e refinando-os de forma que fiquem mais acessíveis e filtrados, tudo depende do intuito da criação da tabela, existem inúmeras transformações para criar-se uma tabela de prata mas a premissa é a mesma, ela é derivada de uma tabela de bronze.

A tabela de Ouro geralmente se dá por alguma agregação entre duas tabelas de prata e novamente, mais refinamento dos dados originários da camada anterior. Nesta camada são hospedados dados o qual já estão prontos para o uso, seja por analistas de BI, seja por trabalho de *Machine Learning*.

Dito isso, através da arquitetura *Multi-hop* tudo se torna interligado entre as tabelas bronze, prata e ouro, uma não existindo sem a anterior. Assim cria-se um fluxo de dados escalonável, podendo ser incrementado e aprimorado indefinidamente. Complementando, mudanças na camada bronze refletirão em todas as tabelas posteriores, criando-se a necessidade de reprocessamento, este que, se feito de forma precária pode resultar em um elevado custo de produção e necessidade de re-trabalho.

Figura 1 – Arquitetura Multi-Hop para Delta Lakes



Fonte: (HEINTZ; LEE,)

2.1.3.1 Delta Table

Utilizadas nos *Delta Lakes*, as tabelas delta são ótimas em melhorar performance e escalabilidade de sistemas de dados. Isso se deve a capacidade das tabelas delta de realizarem *Merges*, mantendo linha das atualizações em tabelas, trabalhando e dando carga apenas em dados novos providos pelo processo ETL. (DELTA. . . ,)

2.1.3.2 Delta Live Tables

Segundo (DATABRICKS. . . , b), *Delta Live Tables* é um *framework* para a construção de pipelines de processamento de dados confiáveis, sustentáveis e testáveis. Assim, as *Delta Live Tables* são um tipo de tabela delta que pode ser atualizada em tempo real, recebendo da fonte de *streaming* o carregamento direto dos dados.

As *Delta Live tables* são modeladas de forma a suportar *streams* de dados de alta velocidade e volume, fornecendo a capacidade de analisar dados de *streaming* em tempo real. Essas tabelas suportam diversas fontes de dados, incluindo *Kafka*, *Azure Event Hub* e *AWS Kinesis* e podem ser acessadas em tempo real utilizando *Spark SQL*, a **API** de *Dataframes* e demais *frameworks* de processamento de dados.

Segundo (L'ESTEVE, 2022) as vantagens de usar*Delta Live Tables* são várias. O **DLT** é capaz de criar gráficos de linhagem de dados, facilitando o rastreamento de status de pipelines com diversas fontes e saídas. Clicando nas tabelas, é possível obter informações de *Schema* e metadados da tabela. Inadequações quanto a qualidade de dados são facilmente rastreáveis com a interface das **DLT**. O **DLT** também suportam cargas incrementais, através da interface é possível atualizar seletivamente as tabelas, atualizando a ingestão sob-demanda ou de forma programada. A interface também mantém registros de eventos dentro da interface. A ferramenta também provém atualizações da tabela em tempo real, monitoramento autônomo, recuperação de versões anteriores, visualização de execuções anteriores e também é capaz de suportar múltiplos ambientes, como desenvol-

vimento e produção. Junto a isso tudo, as **DLT** podem utilizar todas as funcionalidades herdadas do *Databricks*, tal como *time-travel*.

As *delta live tables* também podem ser integradas com o *Auto-Loader* para atualização das tabelas de *streaming*, se mostrando uma solução, apesar de complexa em primeira instância, extremamente eficaz.

Atualmente não existe nenhum trabalho de pesquisa diretamente relacionado a isso e esse é parte do objetivo do trabalho

2.2 Soluções de Armazenamento

2.2.1 Data Warehouse

Um *Data Warehouse* é um banco de dados relacional especializado e de grande escala utilizado para armazenamento, geração de relatórios e análise de dados. O sistema é projetado para lidar e processar grandes quantidades de dados de várias fontes, como sistemas transacionais, arquivos de *log* e fontes externas de dados, e armazená-los em um formato otimizado para consultas e relatórios. Os *data warehouses* permitem que as organizações centralizem seus dados em um único local, facilitando o acesso, análise e relatórios de dados de múltiplas fontes, e apoiam a tomada de decisão e planejamento estratégico.(KIMBALL; ROSS, 2013)

2.2.2 Data Lakes

Segundo (SINGH, 2019) um *data lake* é um repositório centralizado de dados capaz de armazenar uma multitude de dados, variando desde dados estruturados ou semi-estruturados até dados completamente desestruturados. Diante disso, um *data lake* é capaz de armazenar de forma segura, quaisquer tipo de dado, independente de volume ou formato, sendo infinitamente escalável e provendo uma forma mais rápida de análise de *datasets* do que métodos tradicionais.

2.2.3 Data Lakehouse

Um *Data Lakehouse* é definido como um sistema de gerenciamento de dados baseado em baixo custo e armazenamento com acesso direto que também provém gerenciamento analítico de um **DBMS** e funcionalidades estruturais de performance, tais como transações **ACID**, versionamento de dados, auditoria, indexação, *cacheing* e otimização de *queries*. Assim, é possível dizer, que, um *Data Lakehouse* combina as melhores partes de um *Data Lake* e um *Data Warehouse* em uma única solução. Essa solução sendo, baixo custo em um formato aberto e acessível por uma diversidade de sistemas de um *Data Lake*

e as capacidades de otimização e gerenciamento de dados de um *Warehouse*.(ARMBRUST et al.,)

2.2.4 Apache Kafka

O *Apache Kafka* é uma plataforma de *streaming* distribuída, usada para criar pipelines de tempo real e aplicações de *streaming*. O *Kafka* é modelado de forma a suportar *streams* de alta performance, volume e baixa latência, sendo utilizado para processar, armazenar e analisar dados de *streaming* em tempo real.(KAFKA. . . ,)

A plataforma é capaz de suportar grandes quantidades de dados, sendo capaz de processar milhões de eventos por segundo, tudo isso com uma baixa latência.(KAFKA. . . ,)

O Kafka possui diversos casos de uso, dentre eles estão:

- Pipelines de dados em tempo real
- Arquiteturas orientadas a eventos
- Agregação de Registros

Em junção a todas as capacidades citadas, o Kafka também é altamente customizável, podendo ser integrado com diversas ferramentas, tais como *Apache Storm, Spark e Hadoop*, se mostrando um aliado implacável na construção de pipelines de dados.(KAFKA. . . ,)

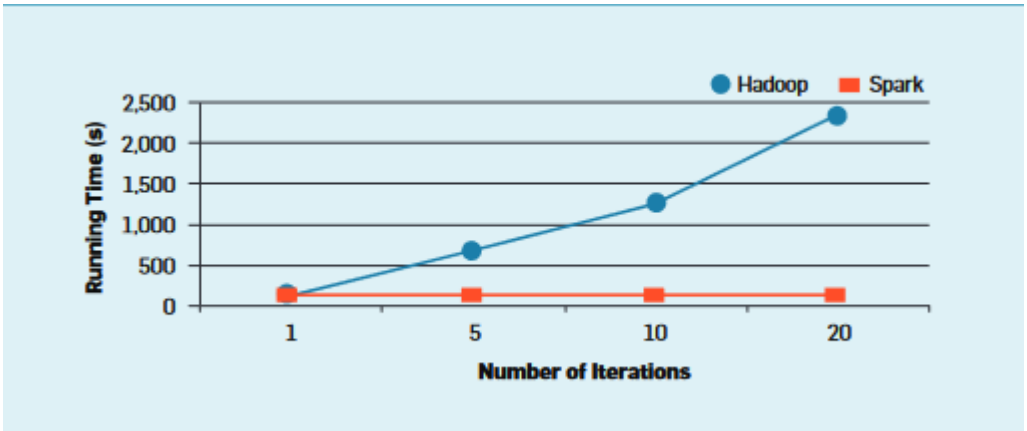
2.3 Apache Spark

O *Apache Spark* é um *framework* de processamento de dados em larga escala que permite aos usuários processar e analisar grandes quantidades de dados de maneira rápida e eficiente. Ele foi desenvolvido pelo *Berkeley Data Analytics Stack (BDAS)* e é baseado no modelo de programação *MapReduce*.

O *Spark* extende o modelo de programação *MapReduce* com uma abstração de compartilhamento de dados chamada de *Resilient Distributed Datasets* ou *RDDs*. Através desta extensão, o *spark* é capaz de capturar uma vasta gama de cargas de processamento que antes necessitariam de motores de processamento separados, tais como *SQL, Streaming, Machine Learning* e processamento de grafos.(ZAHARIA et al., 2016)

O *Spark* por ser um *framework* generalista, possui alguns benefícios em seu uso. Primeiramente, o *Spark* permite desenvolvimento de aplicações com maior facilidade devido a sua API Unificada. O *framework* também é mais eficiente no quesito de combinar tarefas de processamento, onde, o *spark* é capaz de rodar diversas operações sobre um

Figura 2 – Performance de uma regressão logística em Hadoop MapReduce vs. Spark para 100GB de dados em 50 máquinas do m2.4xlarge EC2.



(ZAHARIA et al., 2016)

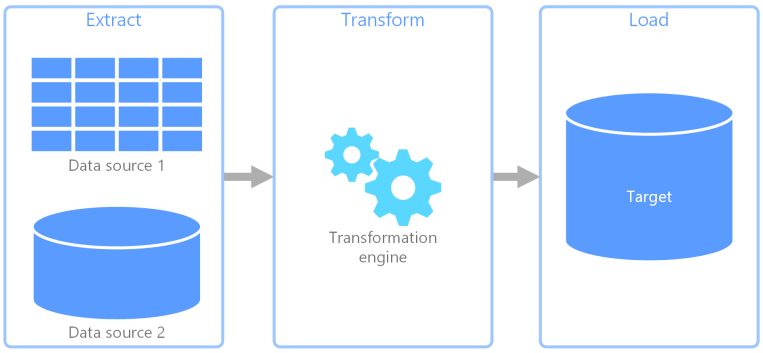
mesmo conjunto de dado, tudo em memória. O *spark* também permite novas aplicações que utilizam tecnologias híbridas, tal como *queries* interativas em um grafo.(ZAHARIA et al., 2016)

Uma das principais vantagens no uso do *Apache Spark* é sua alta velocidade. O *framework* é capaz de processar dados em larga escala em velocidades significativamente mais rápidas do que outros *frameworks* de processamento de dados como o Pandas ou *Hadoop*.(ZAHARIA et al., 2016) Outra vantagem do *Apache Spark* é sua capacidade de trabalhar com vários tipos de dados. Ele é capaz de lidar com dados estruturados, não estruturados e semi-estruturados de maneira eficiente.(SPARK. . . ,)

O *Spark Structured Streaming* é uma funcionalidade do *Apache Spark* que permite aos usuários processar e analisar dados em tempo real de maneira eficiente. Ele é baseado no modelo de programação de fluxo de dados e permite que os usuários trabalhem com dados em fluxo, como dados de *streaming* em tempo real.

O *Spark Streaming* implementa processamento de *stream* incremental utilizando um modelo chamado *discretized streams*. Para implementar *streaming* com o *Spark*, é necessário dividir os dados de entrada em pequenas levas de dado, que são regularmente combinadas com o conjunto de estado armazenado dentro das **RDDs** para produzir novos resultados. Utilizar *streaming* desta forma permite que a recuperação em caso de falha seja menos custosa e também torna possível a combinação de *streaming* com ingestões em *Batch* e também *queries* interativas.(ZAHARIA et al., 2016)

Figura 3 – Representação gráfica de um processo ETL



Fonte: (NAEEM, 2022)

2.4 ETL

ETL é a sigla para **"Extract, Transform and Load"**, significando Extrair, transformar e carregar. Processos de ETL geralmente são atrelados a popular um *Data Warehouse*. Dito isso, os processos de ETL são responsáveis pela extração do dado da fonte, realizando seu transporte para uma área de *Staging*, onde as transformações serão feitas, então, executa-se a transformação ou refinamento desses dados da origem, isso de forma que o dado fique adequado a formatação do **DW**, ainda na transformação, é feito o isolamento de tuplas problemáticas, onde que, através destas, quebras nas regras estruturais do **DW** podem ocorrer. Por fim é realizado o carregamento do dado transformado para a relação devida no armazém, onde, engenheiros, analistas e cientistas poderão acessar os dados.(VASSILIADIS, 2009)

Segundo Kimball, extração é a primeira etapa do processo de inserir dados em um *Data Warehouse*. Nesta etapa é feita a extração dos dados e o carregamentos destes no sistema ETL para manipulação futura. Após a extração partimos para a transformação, nesta etapa limpamos os dados, juntamos dados de origens diversas e de-duplicamos os dados. Finalmente estruturamos os *schemas* e carregamos os dados para os modelos dimensionais.(KIMBALL; ROSS, 2013)

3 Materiais e Métodos

3.1 Considerações Iniciais

A priori, definiu-se um objetivo de pesquisa. O objetivo de pesquisa, que é, construir um **ETL** de *Streaming* utilizando o *Databricks Auto-Loader* e *Delta Live Tables* para ingerir dados em uma solução *Data Lakehouse* e comparando-a com uma abordagem *Spark-Kafka* Tradicional, serviu como pilar para estabelecer campo para o que seria pesquisado e executado durante a execução deste projeto. Diante disso, estabeleceu-se todo o referencial teórico e um plano de execução metodológico para as etapas do projeto, que será detalhado neste capítulo.

3.2 Plano Metodológico

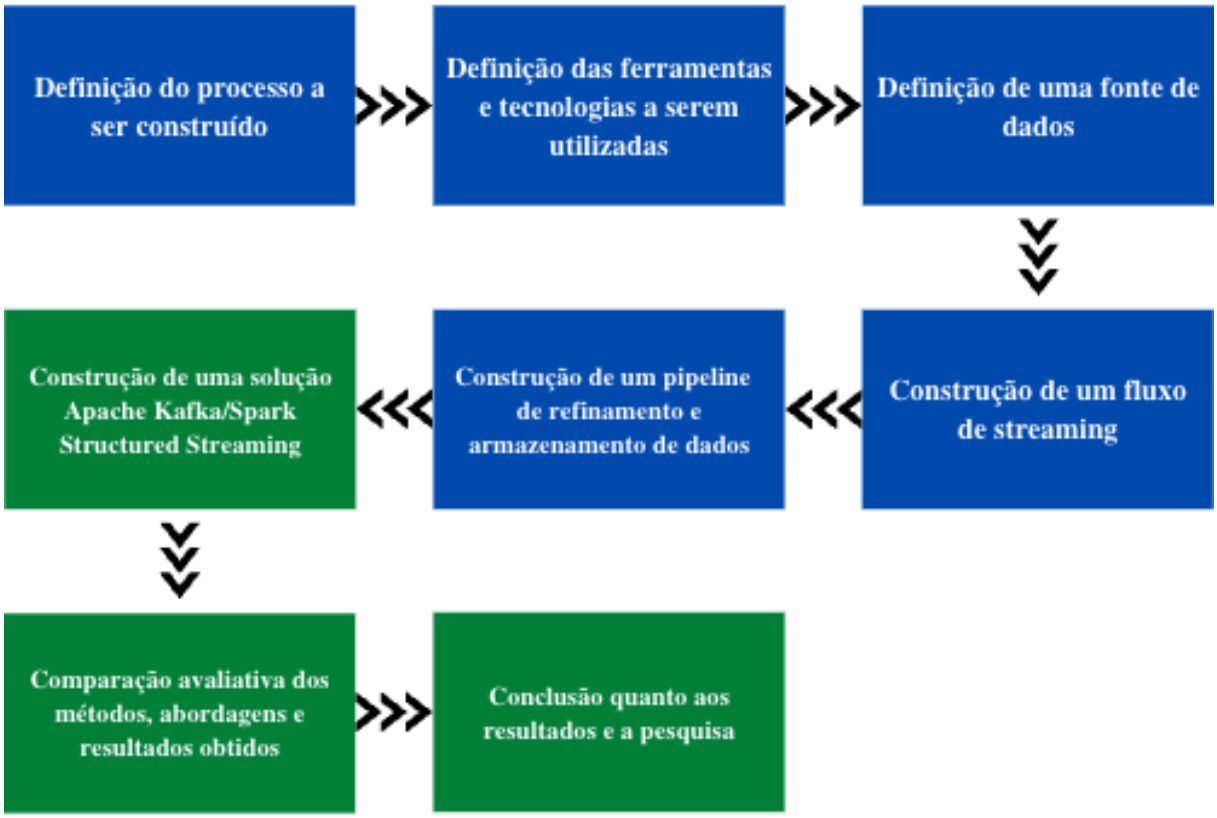
O plano metodológico é composto por 8 etapas:

- Definição do processo a ser construído;
- Definição das ferramentas e tecnologias a serem utilizadas;
- Definição de uma fonte de dados;
- Construção de um fluxo de *Streaming*;
- Construção de um Pipeline de refinamento e armazenamento de dados utilizando a fonte de *Streaming* construída;
- Construção de uma solução *Apache Kafka/Spark Structured Streaming*;
- Comparação avaliativa dos métodos, abordagens e resultados obtidos;
- Conclusão quanto aos resultados e a pesquisa;

3.2.1 Definição do processo a ser construído

Nesta etapa é definido qual será a direção tomada pelo projeto, estabelecendo uma grande área e um sub-área dentro desta. Assim, é necessário encontrar um processo o qual a pesquisa possa ser realizada e gradativamente aprimorada, vindo a ser elaborado no projeto.

Figura 4 – Fluxograma do Plano Metodológico



Fonte: Autoral

3.2.2 Definição das Ferramentas e Tecnologias a serem utilizadas

Durante essa parte do processo, são definidas as tecnologias a serem utilizadas na concepção do processo que foi definido na etapa anterior. Assim, estas tecnologias servirão de alicerce para a construção dos processos, sendo o processo elaborado de acordo com as necessidades das tecnologias definidas.

3.2.3 Escolhendo Uma Fonte de Dados

A seleção de uma fonte confiável e acessível de dados é crucial para a implementação bem-sucedida do projeto proposto. Sem uma fonte de dados confiável e acessível,

a criação de um fluxo de dados se torna inviável, impedindo assim a construção de um pipeline eficaz.

Dito isso, nossa fonte de dados precisa cumprir alguns pre-requisitos:

- A Fonte fosse compatível com o *Databricks*.
- A fonte possui capacidade de fornecer dados consistentemente.
- A fonte fosse aberta, gratuita e de livre acesso.
- A fonte não podia estar relacionada a dados financeiros
- A fonte fosse capaz de oferecer dados que pudessem ser refinados.

3.2.3.1 Event Sim Stream Simulator

O *Event Sim Stream Simulator* é um software capaz de gerar *streaming* de dados falsos sobre músicas. Esta fonte é um software que quando rodado pode ser capaz de gerar dados customizáveis na intensidade e quantidade necessária e solicitada pelo usuário. A fonte pode ser configurada e personalizada a gosto e por não ser real, evitaria quaisquer problemas que poderiam ser originários de uma fonte real.

3.2.3.2 Twitter API

A API do Twitter é uma fonte tradicional de dados e foi considerada como uma alternativa secundária para o projeto. Esta é capaz de fornecer informações relevantes sobre *tweets* de acordo com as seleções e filtros estabelecidos pelo usuário. A **API** retorna os dados dos *tweets* em tempo real, no formato **JSON**, o que requer um processo de extração e análise antes de ser adicionado às tabelas de maneira apropriada.

3.2.4 Construção de um fluxo de streaming

A construção de um *stream* de dados é uma parte crucial da etapa atual do projeto. Para isso, será necessário utilizar a fonte de dados selecionada na etapa anterior. É importante que este *stream* seja capaz de armazenar os dados obtidos em um diretório, permitindo que eles possam ser utilizados futuramente. Além disso, o *stream* deve ser capaz de gerar dados de forma contínua e ininterrupta, para que seja possível obter análises em tempo real sobre o dado processado. É crucial que o *stream* seja configurado de forma adequada para garantir o correto funcionamento do pipeline.

3.2.5 Construção de um pipeline de refinamento e armazenamento de dados

Nesta fase, estruturamos o banco de dados criando tabelas para armazenar os dados e suas operações usando a Arquitetura *Multi-Hop* descrita no referencial teórico.

Além disso, configuramos o processo de ingestão de dados para direcionar dados em tempo real para as tabelas Delta Live criadas.

3.2.6 Construção de uma solução Apache Kafka/Spark Structured Streaming

Neste projeto, adotamos a abordagem tradicional com *Apache Kafka e Spark Structured Streaming*. Construiremos um sistema *stream-kafka-spark*, onde a *stream* de dados alimenta o Kafka que, por sua vez, serve como intermediário para o *Spark* processar os dados e armazená-los em tabelas em um *Data Lake* ou até mesmo em um banco de dados convencional.

3.2.7 Comparação avaliativa dos métodos, abordagens e resultados obtidos

Para execução desta etapa, é necessário definir alguns parâmetros pelos quais serão comparadas as abordagens, os parâmetros são os seguintes:

1. Facilidade de configuração e manutenção: A facilidade de configurar e manter a plataforma de streaming é uma consideração importante.
2. Escalabilidade: A capacidade de lidar com fluxos de dados em larga escala e aumentar/diminuir a escala de acordo com a demanda é uma consideração importante.
3. Tolerância a falhas: A tolerância a falhas é crucial para o processamento de streaming, pois até mesmo pequenas interrupções no fluxo podem causar problemas significativos.
4. Latência: A latência é um parâmetro importante ao processar dados em tempo real.
5. Integração com outras ferramentas: A integração com outras ferramentas de processamento de dados e análise pode ser uma consideração importante.
6. Segurança e conformidade: Os requisitos de segurança e conformidade devem ser considerados ao escolher uma plataforma de processamento de streaming.
7. Custo: O custo da plataforma, incluindo licenças, infraestrutura e custos de manutenção, também deve ser considerado.

Diante disso, ao considerar os parâmetros citados, será possível comparar de forma adequada as execuções e entender em qual parâmetro cada uma se sobressai na construção de um streaming de dados.

3.2.8 Conclusão quanto aos resultados e a pesquisa

Nesta pesquisa, analisaremos a complexidade, usabilidade, performance e custo das soluções desenvolvidas pela *Databricks* e compararemos com o que existe no mercado atualmente. Apresentaremos os resultados e o processo que nos permitiram chegar às conclusões determinadas.

3.3 Referências

As referências utilizadas neste projeto foram selecionadas conforme a evolução do projeto. Quando necessário, buscou-se fontes relacionadas a temas ou ferramentas já definidas para fornecer um embasamento teórico para o autor e o projeto. As referências incluem fontes diversas, como sites e blogs, livros e artigos científicos.

3.4 Considerações Finais

Neste capítulo, apresentamos a metodologia de pesquisa e as estratégias adotadas para desenvolver o projeto, incluindo objetivos e fundamentos teóricos. No capítulo seguinte, faremos uma apresentação detalhada dos resultados alcançados até o momento.

4 Resultados e Discussão

4.1 Definição do Processo a Ser Construído

O Processo foi definido de forma a abordar alguns grandes temas. O primeiro foi a área de engenharia de dados, dentro desta área então afunilou-se as possibilidades chegando a área de *streaming* de dados. Com o *Streaming* de dados como objeto principal definido, viu-se a oportunidade de construir uma solução com base nas tecnologias proprietárias do *Databricks*, estas sendo voltadas para o processo de *streaming*. Contudo, devido ao escopo inicialmente pequeno, uma segunda proposta foi levantada a qual seria a construção de um pipeline de dados, além daquele construído no *Databricks*, agora sendo construído com as ferramentas da *Apache*, o *Kafka* e o *Spark*. No final desse processo, todo o conteúdo seria comparado para entender os benefícios da nova abordagem e como esta se compara a abordagem mais tradicional.

4.2 Definição das ferramentas e tecnologias a serem utilizadas

O projeto requer uma definição clara dos princípios sobre os quais a solução de pesquisa será baseada. Para isso, dentre as diversas opções disponíveis na área de Engenharia de Dados, foram selecionadas algumas ferramentas específicas para serem empregadas nas soluções desenvolvidas neste projeto.

4.2.1 Linguagem de Programação

No desenvolvimento de um projeto na área de engenharia de dados, são utilizadas algumas linguagens padrão para solucionar os problemas encontrados na área, tais como*Python, Scala e SQL*. Neste caso específico, optou-se por utilizar o *Python* como linguagem de programação para a construção dos *streams* de dados, e o *SQL* para criar tabelas para armazenamento dos dados coletados a partir destes *streams*.

4.2.2 Soluções de Armazenamento e Gestão de Dados

Para escolher uma solução de armazenamento de dados, é fundamental conhecer as características dos dados que serão armazenados. Neste caso, estamos lidando com dados semi-estruturados, portanto, uma solução de *Data Warehouse* pode não ser a melhor opção em comparação a um *Data Lake*. No entanto, o *Data Lakehouse* integrado ao *Databricks* oferece a vantagem de combinar os recursos de ambos, tornando-se uma solução completa. Assim, para a primeira fase deste projeto, optamos por utilizar o *Data Lakehouse* integrado

ao *Databricks*. No entanto, para a segunda etapa do projeto, consideramos a possibilidade de construir um *Data Lake* caso seja necessário.

4.2.3 Databricks

A escolha do *Databricks* como objeto de estudo foi motivada pela sua jovialidade no mercado de tecnologia e pela escassa disponibilidade de pesquisas na área. A plataforma oferece ferramentas proprietárias inovadoras e uma abordagem única de integração do processo de *streaming* de dados em uma só plataforma, tornando-a uma boa opção para o projeto. Sendo assim, o *Databricks* foi escolhido como o alvo principal do projeto.

4.2.4 Apache Spark

O *Apache Spark* fornece aos usuários uma vasta gama de ferramentas para trabalho com *streaming* de dados. O *Spark Structured Streaming* é o módulo vital que permite esse processamento de *streams* por parte do *Spark*. Hoje o *Spark* é o padrão na indústria de Big Data e será utilizado neste projeto para processamento de nossa *stream* de dados.

4.3 Definição de uma fonte de dados

4.3.1 EventSim Stream Simulator e Databricks Autoloader

Não foi possível utilizar o *Eventsim Simulator* junto com o *Databricks Autoloader* devido a uma restrição na plataforma. Para a realização da simulação, era necessário instalar uma aplicação. Embora tenhamos tentado algumas soluções alternativas, como rodar a aplicação na linha de comando ou pelo próprio notebook, infelizmente não foi possível estabelecer uma conexão entre o *Eventsim Simulator* e a instância do *Databricks*. Desta forma, a simulação não pôde ser realizada e a conexão com o *Cluster* Interativo disponibilizado pela plataforma não foi possível.

4.3.2 Twitter API

A escolha da API do Twitter para a obtenção dos dados foi uma alternativa viável, uma vez que permitiu coletar os dados de forma real-time e ainda ofereceu recursos que contribuíram para o desenvolvimento da solução proposta. A utilização da API foi fundamental para o sucesso do projeto, permitindo obter uma grande quantidade de dados em tempo real, bem como analisá-los e armazená-los para fins posteriores.

4.4 Estruturando o Processo de Streaming

Inicialmente, com a abordagem do Twitter, viu-se a necessidade de realizar um cadastro como *developed* no Twitter, obtendo assim, diversas chaves para poder utilizar a API, dentre estas, utilizamos a '*bearer_token*', esse *token* provém acesso a consultas por via da *API*.

Para trabalhar o processo de *Streaming* fizemos uso do *Tweepy*, uma biblioteca em *Python* que faz uso da API do Twitter. Junto ao *Tweepy*, utilizou-se o *jsonpickle* e o *colorama* para estruturar o processo de *streaming*.

No processo, definimos inicialmente um diretório onde os dados recebidos seriam depositados, usaremos esses dados para a criação das *delta live tables* futuramente. É nesse diretório onde os arquivos **JSON** capturados através do processo de *streaming* são armazenados e consequentemente, nesse diretório onde os novos dados serão lidos para serem ingeridos nas tabelas que criaremos a seguir.

A V2 da *API* do Twitter não retorna todas as colunas que poderiam ser obtidos de um *Tweet* de forma *automatica*, assim ao solicitar o *tweet* é necessário passar no chamado da *API* os parâmetros solicitados. Dentre os parâmetros solicitados na chamada da *API* estão, a língua, geo localização, o id do autor, o id da conversa, a data de criação, *tweets* mencionados, configurações de resposta, fonte, usuário a ser respondido, métricas não públicas, métricas orgânicas e métricas públicas.

Esses atributos são todos carregados em uma Tabela *Bronze* no *Databricks*, tabela esta que recebe a ingestão de forma crua, sem alguma transformação e posteriormente é refinada para se torna uma tabela *Silver* (prata). Eventualmente, quando os dados forem agregados, a tabela pode se tornar uma tabela *Gold* (ouro), que é a última camada de refinamento antes dela ser disponibilizada para analistas e engenheiros de *Machine Learning*.

Continuando, criou-se um sistema de escrita dos *tweets*, onde, a contagem de *tweets* seria feita a cada *tweet* e a cada 5 *tweets*, estes são escritos em um arquivo '.json', na pasta citada acima. Assim, a cada 5 *tweets*, a função de escrita é chamada, o json é escrito e armazenado no diretório. O '*jsonpickle*' é utilizado para este procedimento.

A *API* conseguiu captar na última execução anterior à escrita deste texto, em aproximadamente 36 minutos de execução, um total de 1126 *Tweets* originários da *Stream*. O conteúdo da *Stream* pode ser alterado sob-demanda, escolhendo diversas palavras chaves e idiomas para os *tweets* captados. No caso deste trabalho, a *string* de seleção foi a seguinte:

```
"(Databricks OR Kafka OR OpenGPT OR GPTChat OR openAI) (lang:en OR lang:pt)"
```

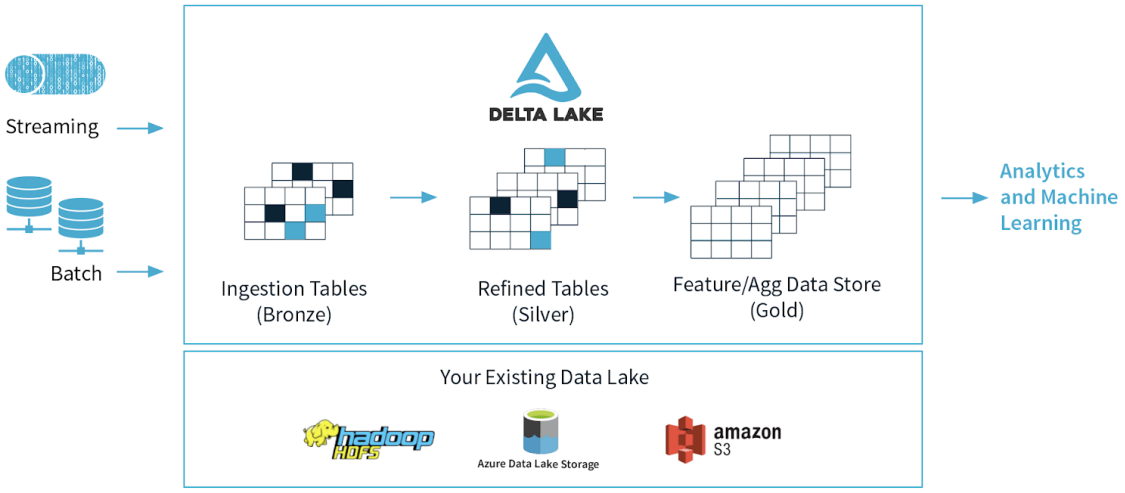
Com essa foi possível obter uma quantidade razoável de dados e relativamente

relacionadas as áreas de pesquisa.

4.4.1 Construção de um Pipeline de refinamento e armazenamento de dados

Usaremos nesta secção do projeto a Arquitetura *Multi-Hop*, explicada na secção do referencial teórico deste trabalho, como já dito, esta arquitetura é composta de três camadas de dados, sendo estas a *Bronze*, *Silver* e *Gold*, onde com a progressão destas, a qualidade do dado é aprimorado.

Figura 5 – *SQL* para criação da tabela *bronze*, utilizando o *Databricks Auto-Loader*



Fonte: (HEINTZ; LEE,)

Diante disso, criamos, para nossa tabela bronze um tabela simples utilizando *SQL* e o *Databricks Auto-Loader*. O *Auto-Loader* é utilizado com a definição do sistema de *cloudFiles*, onde o carregamento será feito automático conforme a inserção de novos dados na pasta 'twitter_bp_2022'. O código em *Spark SQL* utilizado para criação da tabela Bronze pode ser visto na figura 6.

Figura 6 – SQL para criação da tabela bronze, utilizando o *Databricks Auto-Loader*

```
1  CREATE OR REFRESH STREAMING LIVE TABLE tweets_bronze
2
3  AS SELECT * FROM cloud_files(
4    "dbfs:/FileStore/thiago/twitter_bp_2022", "json", map('cloudFiles.inferColumnTypes','false')
5  )
```

Fonte: Autoral

O esquema da tabela pode ser observado na tabela 1.

Tabela 1 – Schema da tabela Bronze.

Atributo	**Tipagem de Dado**
attachments	string
author_id	string
context_annotations	string
conversation_id	string
created_at	string
data	string
edit_controls	string
edit_history_tweet_ids	string
entities	string
geo	string
id	string
in_reply_to_user_id	string
lang	string
non_public_metrics	string
organic_metrics	string
possibly_sensitive	string
promoted_metrics	string
public_metrics	string
referenced_tweets	string
reply_settings	string
source	string
text	string
withheld	string
rescued_data	string

Fonte: Autoral

Com esses dados é possível criar novas tabelas contendo informações que filtrarão os dados da tabela de bronze. Assim, foram criadas três tabelas de prata nessa concepção inicial.

A primeira tabela armazena os *tweets* com a língua em inglesa segunda, com os *tweets* em português e a terceira a contagem da quantidade de *tweets* em cada língua, os códigos podem ser vistos nas imagens 4.4.1, 4.4.1 e 4.4.1:

As tabelas tweets_languages e tweets_pt serão usadas para a criação de uma tabela dourada, contendo a agregação das duas. O SQL pode ser visualizado na figura 4.4.1.

Com base nas tabelas criada, podemos ver o pipeline representado na interface das *Delta Live Tables* na figura 12.

Na figura 9, está a representação do pipeline de dados construído, abrangendo

Figura 7 – Código SQL para a tabela de prata ‘tweets_en‘, contendo os tweets captados em inglês.

```sql
create or replace streaming  live table tweets_en

(constraint valid_language expect (lang == "en") on violation drop row,
constraint valid_id expect (id != "") on violation drop row)

comment 'tabela para textos em inglês'

as
  select id, lang, text from  stream (live.tweets_bronze)
```

Código SQL para a tabela de prata ‘tweets_en‘, contendo os tweets captados em inglês.

Figura 8 – Código SQL para a tabela de prata ‘tweet_pt‘, contendo os tweets captados em português.

```sql
create or replace streaming  live table tweets_pt

(constraint valid_language expect (lang == "pt") on violation drop row,
constraint valid_id expect (id != "") on violation drop row)

comment 'Tabela para textos em português'

as
  select id, lang, text from  stream (live.tweets_bronze)
```

Fonte: Autoral

a tabela de entrada tweets_bronze, que serve de fonte para a criação das tabelas de prata *tweets_en, tweets_languages e tweets_pt.* Com base nas tabelas *tweets_languages e tweets_pt*, é criada a tabela dourada *tweets_language_count.* Também podemos ver a quantidade de registros da execução específica representada na imagem, sendo que, a tabela *tweets_language_count* possuiu, nessa ocasião, 119 registros atualizados.

Figura 9 – Código SQL para a tabela de prata 'tweets_languages', a contagem para cada linguagem dos tweets captados em streaming.

```sql
-- CREATE OR REPLACE LIVE TABLE has same semantics as CREATE LIVE TABLE
create or replace  streaming live table tweets_languages

comment 'Tabela contendo a contagem da quantidade de tweets representantes de determinada língua'

as
select lang, count(*) as count from  stream (live.tweets_bronze) group by lang order by count
```

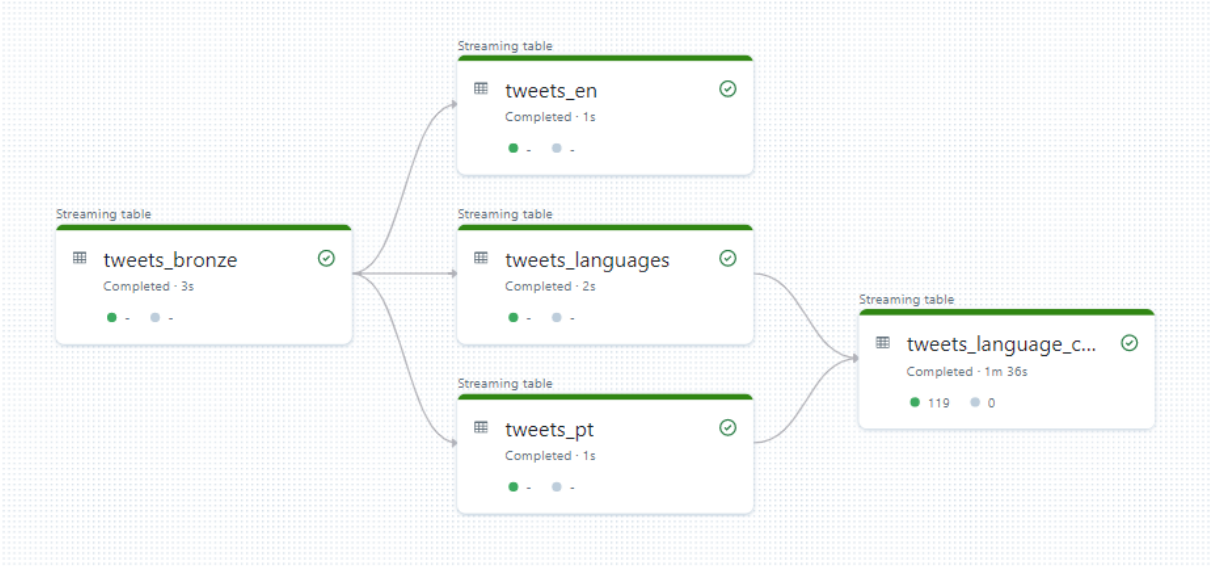
Fonte: Autoral

Figura 10 – SQL para a criação da tabela dourada tweets_language_count, sendo comporta por tweets_pt e tweets_languages.

```sql
create or replace  streaming live table tweets_language_count
as
select id
,      pt.lang
,      text
,      count
from live.tweets_pt as pt
join live.tweets_languages as ln on pt.lang = ln.lang
```

Fonte: Autoral

Figura 11 – Pipeline de dados representado pela interface das delta live tables



Fonte: Autoral

4.5 Construção de uma solução Apache Kafka/Spark Structured Streaming

O processo de estruração do *Kafka* para *streaming* consistiu em algumas etapas:

- Provisionamento de ambiente
- Criação de tópico
- Criação do Producer
- Criação do Consumer

4.5.1 O Provisionamento do Ambiente

Recentemente o *Kafka* atualizou-se para que haja uma integração entre o cliente *Kafka* e o seu auxiliar, o *Zookeeper*. Dito isso, não é mais necessário fazer a configuração de cada um individualmente, sendo necessária a execução única e exclusivamente do *Kafka* para iniciar o ambiente.

Dito isso, boa parte das documentações existentes não levam isso em consideração, tendo em vista que é uma atualização recente, isso acabou por se mostrar problemático na hora de configurar o ambiente.

Com o progresso no uso da ferramenta, viu-se como ótimo utilizar um contêiner *Docker*, utilizando a imagem da *Bitnami/Kafka* para provisionar o ambiente. Embora a execução dos comandos se mostre ainda mais confusa do que inicialmente, essa abordagem facilita a sincronização com o contêiner do *JupyterLab* e assim, permite uma conexão do código sendo produzido com o *Apache Kafka* através de uma *Docker Network*.

4.5.2 Criação de Tópico

O processo de criação de tópico é relativamente simples, é necessário criar um tópico capaz de receber as mensagens que estamos recebendo e enviando através do nosso *stream* de dados.

Como nosso fluxo de dados não é tão elevado, não é necessário mexer nas configurações básicas de criação de um tópico, usando apenas uma partição nesse caso e criando-o com o seguinte comando:

```
kafka-topics.sh --create --replication-factor 1 --bootstrap-
server localhost:9092 --topic twitter-stream
```

Com este tópico é podemos inserir dados no tópico através de um *Producer* e consumir o dado inserido através de um *Consumer*.

4.5.3 Criação do Producer

O módulo do produtor é responsável por enviar os dados para o Kafka, atuando como uma ponte entre o fluxo de dados e o sistema Kafka. Para estabelecer essa conexão, foi necessário extrair os dados do fluxo de *streaming* e armazená-los em documentos, criando uma pasta que contém um arquivo para cada objeto **JSON** obtido durante o processo de *streaming*.

Embora tenham sido feitas várias tentativas de alimentar diretamente o tópico do *Kafka* a partir do fluxo de *streaming*, surgiram problemas recorrentes que não retornavam erros e eram insolúveis. Esses problemas causavam falhas no fluxo e impediam a alimentação do tópico, levando à adoção de uma abordagem baseada em arquivos.

A ingestão dos arquivos ocorre em uma pasta designada, onde há um produtor pronto para execução. Quando executado, esse produtor carrega as mensagens obtidas para o tópico do *Kafka*. Esse processo é executado em micro lotes, sendo realizado a cada minuto.

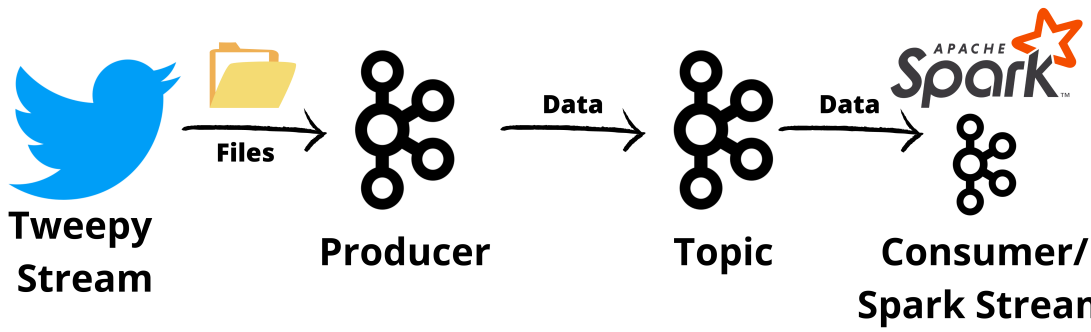
4.5.4 Criação do Consumer

Na aplicação do consumidor, desenvolvemos uma solução para receber e processar os dados provenientes do Kafka. Utilizamos o Spark Structured Streaming, que desempenha um papel fundamental no tratamento dos dados brutos assimilados pela aplicação. Nessa etapa, aplicamos regras de negócio e criamos tabelas conforme necessário.

Em particular, estabelecemos uma tabela de entrada responsável por receber os dados em sua forma bruta, porém, após passarem pela função de parse do Spark, esses dados são inicialmente consumidos pela aplicação no formato de strings. Em seguida, ocorre a transformação desses dados em uma tabela estruturada, com as colunas previamente definidas.

Dessa forma, o Spark Structured Streaming possui a capacidade de manter uma tabela de streaming, que é continuamente atualizada à medida que novos dados são recebidos pelo tópico do Kafka por meio do produtor.

Figura 12 – Pipeline de dados representado a dinâmica dos consumers, producers, a stream e o Spark



Fonte: Autoral

4.6 Comparação avaliativa dos métodos, abordagens e resultados obtidos

4.6.0.1 Facilidade de Configuração e Manutenção

Com relação à facilidade de configuração e manutenção, o *Databricks Auto-loader* é mais fácil de configurar e gerenciar. Este foi projetado para simplificar o processo de ingestão de dados em larga escala em um ambiente de nuvem e minimizar a necessidade de gerenciamento de infraestrutura e configuração complexa. Com o ambiente da *Databricks*, não é necessário provisionar e configurar recursos de servidor, como máquinas virtuais e clusters de computação, ou instalar e configurar softwares de processamento de dados e *streaming*.

Em vez disso, o *Databricks Auto-loader* é integrado à plataforma unificada de análise de dados da *Databricks*, o que significa que os usuários podem usar a interface do usuário familiar e intuitiva da plataforma para configurar e gerenciar a ingestão de dados. Os usuários podem simplesmente fornecer informações básicas, como a fonte de dados, o tipo de dados e a frequência de ingestão, e as *Delta Live Tables* junto ao *Auto-loader* cuidará do restante. As D*elta Live Tables* junto ao *Auto-loader* também oferecem recursos de monitoramento e alerta automatizados para facilitar a manutenção contínua do processo de ingestão de dados.

Talvez um dos principais problemas com relação ao uso das *Delta Live Tables* diz respeito a documentação, não é uma *feature* que está no mercado a muito tempo e sua

disponibilidade é extremamente limitada, desta forma, pouco se encontra sobre as mesmas em buscas online. O que obtive de material sobre se encontra preso sobre o alto custo de um curso de engenharia de dados da *Databricks* ou com os engenheiros que fornecem apoio as equipes que contratam a plataforma.

Por outro lado, a disponibilidade de materiais sobre Apache Kafka beiram ao infinito, é possível encontrar diversos cursos e materiais sobre os mais variados tópicos de uso do Apache Kafka, o mesmo pode ser dito sobre o *Spark Structured Streaming* que possui uma documentação extremamente sólida e apesar de nenhum da configuração de nenhum dos processos ser simples, é claramente mais acessível a um desenvolvedor que intende em se desenvolver em uma das ferramentas de *streaming*.

Complementando, o *Apache Kafka* requer mais conhecimento e esforço para configurar e gerenciar. Para usar o *Apache Kafka*, os usuários precisam primeiro configurar e gerenciar a infraestrutura do *Kafka*, como *brokers* e tópicos, que podem ser complexos de configurar e gerenciar. O *Spark Structured Streaming* também não é nada trivial, por sua abordagem ser de execução contínua, o *troubleshooting* dentro do código se mostra excessivamente complicado em primeiro contato.

Embora o *Apache Kafka* possa oferecer mais flexibilidade e controle sobre o processo de ingestão e processamento de dados, ele também pode ser mais complexo e exigir mais esforço de configuração e manutenção do que as soluções da *Databricks*, muito embora não exista muita documentação sobre, as *Delta Live Tables* se mostram tão complexas quanto a configuração de uma *stream* com o *Spark*. A configuração do *Auto-loader* é infinitamente mais simples do que a de um ambiente *Kafka*.

Portanto, a escolha entre as soluções da *Databricks* e o *Kafka* com *Spark* dependerá em grande parte do nível de habilidade técnica da equipe, se a equipe já possui um ecossistema com o *Kafka* desenvolvido ou tenha apoio da equipe de suporte da *Databricks*, dos requisitos de flexibilidade e controle e dos recursos disponíveis para gerenciar a plataforma.

4.6.1 Escalabilidade

No que diz respeito à escalabilidade, tanto as soluções da *Databricks* quanto o *Apache Kafka* integrado ao *Spark Structured Streaming* são soluções altamente escaláveis e projetadas para lidar com grandes volumes de dados em tempo real.

O Auto-*loader* por fazer parte da *Databricks*, se torna altamente escalável e dimensionável. O uso de *clusters* de computação distribuídos para processar grandes quantidades de dados de forma paralela, pode ser escalado automaticamente para lidar com picos de carga de dados com a *feature* de *auto-scaling*. Além disso, a plataforma da *Databricks* tem compatibilidade com uma diversidade de fontes de dados, como *Amazon S3, Azure Blob*

Storage e Google Cloud Storage, plataformas estas que podem ser escaladas para lidar com grandes volumes de dados.

Com relação ao *Apache Kafka*, este também é altamente escalável e é pode ser utilizado tão bem ou até de forma mais eficaz que o *Auto-Loader*. O *Apache Kafka* é capaz de lidar com fluxos de dados em tempo real em grande escala e pode ser escalado horizontalmente para lidar com picos de carga de dados, tudo isso aliado ao *Apache Spark* que também é altamente escalável e pode ser executado em *cluster* de computadores distribuídos para processar grandes volumes de dados em paralelo.

Em geral, tanto o *Auto-Loader* quanto o *Kafka* dão conta do recado quando o se trata de *streaming*, sendo ambas as soluções altamente escalonáveis e capazes de lidar com *streamings* de alta concorrência, neste sentido, é possível concluir que a escolha quando se trata de concorrência pode ser determinada pelo fator de ter ou não o *Databricks* disponível, porque este, apresenta recursos de computação em nuvem que podem facilmente ser escalonados sob-demanda, o mesmo pode ou não ser verdadeiro para o *Apache Kafka*.

4.6.2 Tolerância a Falhas

Com relação à tolerância a falhas, tanto o *Databricks Auto-loader* quanto o *Apache Kafka* integrado ao *Spark Structured Streaming* são altamente tolerantes a falhas e trabalham de forma a garantir que o processamento de dados não seja interrompido em caso de falhas.

A plataforma da *Databricks* é projetada para ser altamente resiliente e tolerante a falhas, sendo assim, esta oferece recursos de alta disponibilidade e *failover* automático, o que significa que, em caso de falha de um nó ou instância da plataforma, o processamento de dados é transferido automaticamente para outro nó ou instância sem interromper o processo de ingestão de dados. Além disso, a *Databricks* oferece recursos de backup e recuperação de desastres para garantir que os dados sejam protegidos contra perda ou corrupção.

Por outro lado, o *Apache Kafka* também é projetado para ser altamente tolerante a falhas. O *Apache Kafka* é projetado para suportar replicação de dados para garantir que os dados não sejam perdidos em caso de falha de hardware ou software. Além disso, o *Apache Spark* suporta a recuperação automática de falhas, o que significa que, em caso de falha de um nó ou instância do *cluster*, o processamento de dados é transferido automaticamente para outro nó ou instância sem interromper o processo de ingestão de dados.

Em geral, tanto o *Databricks Auto-loader* e as *Delta Live Tables* quanto o *Apache Kafka* utilizando o *Spark Structured Streaming* são soluções altamente tolerantes a falhas e projetadas para garantir a continuidade do processamento de dados em caso de falhas de hardware ou software. Neste caso, a escolha entre as duas soluções dependerá princi-

palmente dos requisitos específicos do projeto e das preferências da equipe técnica, não havendo favorito claro quando se trata de tolerância a falhas.

4.6.3 Latência

Com relação à latência, as *Delta Live Tables* integradas ao *Auto-loader* e o *Apache Kafka* com o *Spark Structured Streaming* possuem características diferentes.

O Databricks *Auto-loader* e as *Delta Live Tables* são projetados para fornecer baixa latência e alta velocidade de ingestão de dados, permitindo que os dados sejam carregados em tempo real em sistemas de armazenamento de dados. Além disso, o recurso de *Delta Live Tables* permite que seja realizado *streaming* de dados em tempo real, possibilitando o processamento imediato dos dados após a ingestão.

Por outro lado, o *Apache Kafka* é projetado para ser altamente escalável e tolerante a falhas, mas pode ter uma latência um pouco maior do que o *Databricks Auto-loader* e as *Delta Live Tables*. Isso ocorre porque o *Apache Kafka* usa um modelo de mensagens chamadas de *micro-batches*, o que significa que as mensagens são agrupadas e processadas em lotes em vez de serem processadas individualmente em tempo real. No entanto, o *Apache Kafka* ainda é capaz de lidar com grandes volumes de dados em tempo real com latências muito baixas em comparação com outras tecnologias de *streaming*.

Em geral, o *Databricks Auto-loader* é mais adequado para casos de uso em que a baixa latência é crítica.

4.6.4 Integração com outras ferramentas

Com relação à integração com outras ferramentas, tanto as soluções da *Databricks* quanto o *Apache Kafka* com *Spark* são facilmente integrados com outras ferramentas e tecnologias de *Big Data*.

A Databricks oferece uma plataforma completa de Big Data que inclui *Apache Spark*, *SQL Analytics*, *Delta Lake* e outras tecnologias. Isso significa que o *Databricks Auto-loader* pode ser facilmente integrado com outras ferramentas da plataforma da *Databricks* para criar soluções de Big Data completas. Além disso, o *Databricks Auto-Loader* é projetado para ser compatível com várias fontes de dados, como bancos de dados relacionais, arquivos **CSV**, **JSON** e outros formatos de arquivo, possibilitando assim, que uma vasta gama de fontes seja integrada também, as *Delta Live Tables*, visto que são ferramentas complementares em muitos casos.

O Apache Kafka por outro lado, pode ser integrado com várias ferramentas e tecnologias de Big Data, incluindo o *Apache Spark*. O Apache Kafka também possui uma **API** de conectores que permite que os dados sejam facilmente transferidos entre o Kafka

e outras ferramentas de Big Data, como bancos de dados **NoSQL**, *data warehouses*, data lakes e outras tecnologias.

Por fim, tanto o *Databricks Auto-loader* e as *Delta Live Tables* quanto o *Apache Kafka* com *Spark* podem ser facilmente integrados com outras ferramentas e tecnologias de Big Data, permitindo que as empresas criem soluções completas de Big Data que atendam às suas necessidades específicas. Mediante isso, a escolha de ferramenta quando se trata de integração, ficará decidida por outros fatores.

4.6.5 Segurança e Conformidade

Com relação à segurança e conformidade, tanto o *Databricks Auto-loader* e as *Delta Live Tables* quanto o *Apache Kafka* com *Spark* oferecem recursos para garantir a segurança e a conformidade dos dados.

O *Databricks Auto-loader* pode ser usado com a plataforma Delta Lake, que fornece recursos de segurança, como criptografia de dados em repouso e em trânsito, controle de acesso baseado em função e auditoria de dados, também fornecendo transações **ACID** compatíveis, o que significa que as transações garantem a atomicidade, consistência, isolamento e durabilidade de cada dado transacionado. Além disso, o *Databricks Auto-loader* é compatível com vários provedores de serviços em nuvem, que também oferecem recursos de segurança para proteger os dados na nuvem.

O *Apache Kafka* usando o *Spark* também possui recursos de segurança para garantir a proteção dos dados. O *Apache Kafka* tem um modelo de segurança baseado em autenticação e autorização, que permite que os usuários controlem o acesso aos dados. O *Apache Spark* possui recursos de segurança, como criptografia de dados em repouso e em trânsito, e autenticação de usuários.

Além disso, ambos o *Databricks Auto-loader* e o *Apache Kafka* utilizando o *Spark* são compatíveis com vários padrões de conformidade, tal como a **LGPD**. Isso significa que as empresas podem usar essas soluções de *streaming* de dados sem comprometer a conformidade com as regulamentações de segurança e privacidade de dados.

Resumindo, tanto o *Auto-loader* quanto o *Apache Kafka* são projetados para serem seguros e conformes com as regulamentações de segurança e privacidade de dados. Não havendo assim, nenhuma disparidade entre a escolha das ferramentas quando se trata de segurança e conformidade.

4.6.6 Custo

Quanto aos custos, tanto as soluções da *Databricks* quanto o *Apache Kafka* com *Spark* podem ter custos variados, dependendo dos recursos necessários para o projeto.

O *Databricks* é um serviço pago, o custo do serviço varia de acordo com a quantidade de dados processados e armazenados e com os recursos computacionais necessários. Além disso, a *Databricks* calcula o custo por uma unidade de medida chamada **SKU**, que é calculada de acordo com o gasto utilizado por processamento, cada *cluster*, dependendo da sua potência, **RAM**, quantidade de *workers* e também se é *cluster* interativo ou apenas para *jobs* é levado em consideração no cálculo do *SKU*, em geral, o valor do gasto de um job-cluster é 1/4 do valor do *cluster* interativo, no nosso caso com as *Delta Live Tables*, utilizamos um *job-cluster*, então o custo é diminuído com relação a outras plataformas de processamento e computação em nuvem.

Já o *Apache Kafka* é uma solução de código aberto, o que significa que é gratuito para download e uso. No entanto, o custo pode aumentar à medida que mais recursos são adicionados para atender às necessidades de processamento de dados. Por exemplo, para executar o *Apache Kafka* em produção, pode ser necessário usar servidores dedicados ou contratar serviços em nuvem para hospedar e gerenciar os *clusters*, isso pode acatar em custos superiores ao da *Databricks*.

Além disso, tanto o *Databricks* quanto o *Apache Kafka* com *Spark* podem exigir recursos adicionais para integrar outras ferramentas e tecnologias de Big Data, o que pode aumentar ainda mais os custos. Neste caso, a instância da ***AWS/GCP/****Azure* por exemplo, tem um custo adicional atrelado ao fornecimento da instância de computação em nuvem.

Em geral, os custos das ferramentas da *Databricks* podem ser mais previsíveis, uma vez que é um serviço pago com custos definidos para uso de recursos e serviços adicionais, enquanto os custos do Apache *Kafka* com *Spark Structured Streaming* podem variar mais dependendo do tamanho e complexidade do projeto, para projetos menores, que não necessitam de computação em nuvem, o custo é zero, para projetos maiores, a solução da *Databricks* pode se mostrar mais custo-eficiente.

4.7 Conclusão quanto aos resultados e a pesquisa

Este trabalho foi elaborado com o objetivo de identificar pontos-chave que tornam uma ferramenta superior ou não a outra e como isso pode influenciar na construção de um processo de *streaming*, levantando diversos aspectos relacionados ao uso de cada uma das ferramentas, sejam eles positivos ou negativos. Isso foi desenvolvido por meio de uma metodologia imposta e detalhada nos métodos e materiais.

Inicialmente, definimos o processo a ser construído, o qual serviu como base para a determinação dos métodos utilizados nesta pesquisa. Definimos as ferramentas que auxiliariam na criação desse processo, estabelecemos uma fonte de dados que serviria como fundamentação para a criação do nosso pipeline de dados. Construímos um fluxo de *stre-*

aming e, com isso, elaboramos um pipeline de refinamento e armazenamento de dados. Além disso, construímos uma solução baseada em *Apache Kafka/Spark Structured Streaming*, realizamos uma avaliação comparativa dos métodos, das abordagens e dos resultados obtidos e, por fim, concluímos este trabalho.

No que diz respeito à definição do processo, abordamos alguns temas relevantes, chegando à determinação do trabalho com *streaming* de dados. Assim, definimos o uso do *Databricks Auto-Loader* e um ponto de pesquisa que consistiria na comparação deste com um método mais tradicional, que foi definido como uma abordagem *Kafka*. Esse processo passou por algumas alterações durante a execução do projeto, com a adição de tópicos necessários e a exclusão daqueles desnecessários para alcançarmos o objetivo final.

Para a definição das ferramentas, restringimo-nos a uma seleção específica, determinada pelo processo de *streaming* que construímos. Essas ferramentas atenderam às necessidades do projeto e pouco foi alterado em relação ao que foi estabelecido inicialmente.

No que diz respeito à fonte de dados, inicialmente optamos pelo uso de uma ferramenta de código aberto que mostrou-se incapaz de atender aos requisitos definidos nos materiais e métodos. Por fim, escolhemos utilizar a API do Twitter, uma das poucas soluções de *streaming* de código aberto disponíveis no mercado.

Com a fonte de dados definida, bastou-nos construir um processo de *streaming* de dados, que recebesse e armazenasse os dados em um sistema de microbatches. Para isso, utilizamos a API do *Twitter* e a biblioteca *python Tweepy*. Dessa forma, com os dados obtidos, foi possível construir todo um pipeline de dados que serviu como base para a análise posterior, comparando a primeira abordagem com a segunda.

Em seguida, desenvolvemos a segunda solução utilizando as ferramentas convencionais. Isso envolveu a provisão do ambiente, que pode ser considerada a etapa mais complexa do processo. Essa etapa incluiu a estruturação lógica, que consistiu na criação de tópicos, bem como a implementação do produtor (*producer*) e consumidor (*consumer*).

Por fim, comparamos ambas as soluções com base naquilo que foi construído, assim, ambas as soluções de *streaming* de dados, as *Delta Live Tables* em junção ao *Auto-loader* e *Apache Kafka* usando o *Spark Structured Streaming*, têm vantagens e desvantagens, dependendo dos requisitos específicos do projeto. Ao comparar essas soluções, é importante considerar uma série de parâmetros, como facilidade de configuração e manutenção, escalabilidade, tolerância a falhas, latência, integração com outras ferramentas, segurança e conformidade, e custo.

Em geral, o Databricks Auto-loader pode ser uma solução mais fácil de usar e gerenciar, com uma abordagem de configuração e manutenção simplificada. Além disso, a plataforma Databricks é conhecida por sua escalabilidade e recursos de segurança ro-

bustos.

Já o *Apache Kafka* com *Spark Structured Streaming* oferece maior flexibilidade e controle sobre o *streaming* de dados, permitindo um processamento de dados em tempo real mais rápido e uma integração mais ampla com outras ferramentas e serviços de *Big Data*. Além disso, como uma solução de código aberto, o *Apache Kafka* e o *Spark* podem se provar uma opção mais econômica, mas pode acabar exigindo mais recursos de gerenciamento e manutenção.

Em última análise, a escolha entre essas soluções dependerá dos requisitos e prioridades específicos do projeto, bem como das políticas de segurança e conformidade da empresa e dos recursos orçamentários disponíveis. Ambas se mostram eficientes em resolver o problema proposto e devem ser levadas em consideração. A solução da Databricks por ser nova ainda precisa trilhar um longo caminho para roubar mercado do Kafka, que já é uma ferramenta consolidada no Mercado.

Diante disso, conseguimos atingir os objetivos específicos do trabalho, que consistiam em construir um *pipeline* de *streaming* utilizando o *Databricks Auto-Loader* e as *Delta Live Tables*, construir um *pipeline* de *streaming* usando o *Kafka* e o *Spark Structured Streaming*, encontrar uma fonte de dados que atinja as necessidades do projeto, modelar o design e a arquitetura do processo de **ETL** desenvolvido e avaliar compratívamente as abordagens utilizadas, todos estes objetivos foram atingidos e com isso, atingimos o objetivo geral deste projeto.

Referências

APACHE Kafka - market share, competitor insights in queueing, messaging and background processing. Disponível em: <https://www.slintel.com/tech/queueing-messaging-and-background-processing/apache-kafka-market-share>. Citado na página 13.

ARMBRUST, M. et al. *Lakehouse: A new generation of open platforms that unify data ...* Disponível em: <https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf>. Citado na página 19.

DATA Lakehouse Architecture and AI Company. 2023. Disponível em: <https://www.databricks.com/>. Citado 2 vezes nas páginas 13 e 15.

DATABRICKS - market share, competitor insights in Big Data Analytics. Disponível em: <https://www.slintel.com/tech/big-data-analytics/databricks-market-share>. Citado na página 13.

DATABRICKS Documentation. <https://docs.databricks.com>. Accessed: 2022-08-12. Citado 2 vezes nas páginas 15 e 17.

DELTA Lake Docs. <https://docs.delta.io/>. Accessed: 2023-01-14. Citado na página 17.

HEINTZ, B.; LEE, D. *Productionizing Machine Learning with Delta Lake*. Disponível em: <https://www.databricks.com/blog/2019/08/14/productionizing-machine-learning-with-delta-lake.html>. Citado 3 vezes nas páginas 16, 17 e 30.

KAFKA Documentation. <https://kafka.apache.org/documentation/>. Accessed: 2022-10-12. Citado na página 19.

KIMBALL, R.; ROSS, M. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. 3rd. ed. [S.l.]: Wiley Publishing, 2013. ISBN 1118530802. Citado 3 vezes nas páginas 13, 18 e 21.

L'ESTEVE, R. *The Azure Data Lakehouse Toolkit*. Apress, 2022. Disponível em: <https://doi.org/10.1007/978-1-4842-8233-5>. Citado 2 vezes nas páginas 15 e 17.

MSSAPERLA. *O que É delta lake? – Azure Databricks*. [s.n.]. Disponível em: <https://learn.microsoft.com/pt-br/azure/databricks/delta/>. Citado na página 16.

NAEEM, T. *Processamento de Dados ETL: Integrações, Processos e componentes explicados*. 2022. Disponível em: <https://www.astera.com/pt/tipo/blog/dados-etl/>. Citado na página 21.

SINGH, A. Architecture of data lake. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, Technoscience Academy, p. 411–414, mar. 2019. Disponível em: <https://doi.org/10.32628/cseit1952121>. Citado na página 18.

SPARK Documentation. <https://spark.apache.org>. Accessed: 2022-07-12. Citado na página 20.

VASSILIADIS, P. A survey of extract–transform–load technology. *Integrations of Data Warehousing, Data Mining and Database Technologies*, p. 171–199, 2009. Citado na página 21.

ZAHARIA, M. et al. Apache spark. *Communications of the ACM*, Association for Computing Machinery (ACM), v. 59, n. 11, p. 56–65, out. 2016. Disponível em: <https://doi.org/10.1145/2934664>. Citado 2 vezes nas páginas 19 e 20.