UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GUILHERME MACEDO DE SOUZA

PIPELINES DE DADOS: UM ESTUDO DE CASO COMPARATIVO ENTRE ETL E ELT USANDO UM DATASET DO FIFA 21

Recife
2025

GUILHERME MACEDO DE SOUZA

PIPELINES DE DADOS: UM ESTUDO DE CASO COMPARATIVO ENTRE ETL E ELT USANDO UM DATASET DO FIFA 21

Trabalho apresentado ao programa de
Graduação em Ciência da Computação do
Centro de Informática da Universidade
Federal de Pernambuco como requisito
parcial para a obtenção do grau de Bacharel
em Ciência da Computação.

Recife
2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Souza, Guilherme Macedo de.
Pipeline de Dados: Um Estudo de Caso Comparativo entre ETL e ELT usando um Dataset do FIFA 21 / Guilherme Macedo de Souza. - Recife, 2025.
77p : il., tab.

Orientador(a): Valéria Cesário Times
Cooorientador(a): Carlos André Guimarães Ferraz
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2025.
Inclui referências, apêndices.

1. ETL. 2. ELT. 3. Conjunto de Dados. 4. Pipeline de Dados. 5. Python. 6. DBT. I. Times, Valéria Cesário. (Orientação). II. Ferraz, Carlos André Guimarães. (Coorientação). IV. Título.

000 CDD (22.ed.)

GUILHERME MACEDO DE SOUZA

PIPELINES DE DADOS: UM ESTUDO DE CASO COMPARATIVO ENTRE ETL E ELT USANDO UM DATASET DO FIFA 21

Trabalho apresentado ao programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Apresentado em: 04/04/2025

Banca Examinadora:

Profa. Dra. Valéria Cesário Times (Orientador)
Universidade Federal de Pernambuco

Prof. Dr. Carlos André Guimarães Ferraz (Examinador Interno)
Universidade Federal de Pernambuco

Recife, 04 de Abril de 2025.

AGRADECIMENTOS

À minha professora e orientadora, Valéria Times, por não apenas ter me guiado pelo árduo caminho do trabalho de graduação, mas também por ter sido a primeira a me apresentar ao mundo dos dados — mundo esse onde hoje atuo profissionalmente,

Aos meus pais, Dona Simone e Seu Gustavo, e ao meu irmão Gustavo, por pavimentarem cada passo da minha jornada com inspiração, incentivo e carinho,

À minha noiva, amiga e companheira, Larissa Maria, que há sete anos ilumina meus dias e me faz o homem mais feliz do mundo,

Aos Porteiros, Hogsmeade, ++, Rolês & Drinks e tantos outros amigos que estiveram ao meu lado nos melhores e piores momentos,

Aos amigos da graduação, que trouxeram leveza, risadas e companheirismo ao dia a dia, tornando os desafios meros detalhes,

À Universidade Federal de Pernambuco e ao Centro de Informática, que, embora tenham me privado de noites de sono e momentos de paz, me acolheram de braços abertos e moldaram o profissional que sou hoje.

"Um passo à frente e você não está mais no mesmo lugar."
— Chico Science, Um Passeio no Mundo Livre.

RESUMO

Esta pesquisa apresenta um estudo de caso comparativo entre as arquiteturas ETL (Extract, Transform, Load) e ELT (Extract, Load, Transform) no contexto de fluxos de dados. Para isso, foram desenvolvidos dois fluxos: um utilizando a arquitetura ETL, onde a transformação dos dados é realizada em Python antes da carga de dados em um banco PostgreSQL, e outro seguindo a arquitetura ELT, em que os dados são carregados diretamente no banco e transformados utilizando a ferramenta DBT (Data Build Tool). O estudo utiliza um conjunto de dados público do jogo FIFA 21, obtido no Kaggle e inicialmente não tratado, para demonstrar as etapas de limpeza, transformação e carga em cada abordagem de extração de dados considerada. Além da análise prática, o trabalho contextualiza historicamente a evolução dessas arquiteturas, discutindo suas aplicações, vantagens e limitações no cenário tecnológico atual. Os resultados evidenciam as diferenças entre ETL e ELT em termos de complexidade, escalabilidade, manutenibilidade e adequação a diferentes cenários, contribuindo para a compreensão das melhores práticas em engenharia de dados.

Palavras-chave: ETL; ELT; Conjunto de Dados; Pipeline de Dados; Python; DBT.

ABSTRACT

This research presents a comparative case study between the ETL (Extract, Transform, Load) and ELT (Extract, Load, Transform) methodologies in the context of data pipelines. For this purpose, two pipelines were developed: one using the ETL approach, where data transformation is performed in Python before loading data into a PostgreSQL database, and the second pipeline approach follows the ELT methodology, in which data is loaded directly into the database and transformed using the DBT technology (Data Build Tool). The study uses a public dataset from the FIFA 21 game, obtained from Kaggle and initially unprocessed, to demonstrate the steps of cleaning, transformation, and loading in each approach. In addition to the practical analysis, the research provides a historical context on the evolution of these methodologies, discussing their applications, advantages, and limitations in the current technological scenario. The results highlight the differences between ETL and ELT in terms of complexity, scalability, maintainability, and suitability for different scenarios, contributing to a better understanding of best practices in data engineering.

Keywords: ETL; ELT; Dataset; Data Pipeline; Python; DBT.

LISTA DE QUADROS

LISTA DE FIGURAS

LISTA DE PSEUDOCÓDIGOS

LISTA DE CÓDIGOS

SUMÁRIO

1 Introdução

1.1 Motivação

A transformação e o gerenciamento de dados são processos essenciais para que a tomada de decisão seja feita de forma eficiente, independente da área de atuação das organizações. O volume de dados gerado cresce de forma exponencial, potencializado pela digitalização de processos e pela Internet das Coisas (IoT) [1]. Essa escalada no volume de dados traz consigo desafios e complexidades em questões como coleta, processamento, armazenamento e análise, tornando cada vez mais crítica a eficiência, solidez e confiabilidade dos fluxos de dados implementados [2].

Diante desse cenário, duas arquiteturas de fluxo de dados destacam-se: ETL (Extract, Transform, Load) [3, 4] e ELT (Extract, Load, Transform) [3, 4]. ETL, uma abordagem tradicional amplamente utilizada desde os primórdios da integração de dados, com o surgimento do Data Warehouse [5], realiza a transformação dos dados antes da carga dps dadps na base destino. Em contrapartida, o ELT, impulsionado pelo advento do Big Data [6] e avanço dos bancos de dados em nuvem e sistemas de processamento paralelo, permite que os dados sejam carregados primeiro e transformados posteriormente dentro do ambiente do banco de dados [1].

O estudo apresentado nesse documento busca adentrar nessa discussão ao apresentar um estudo de caso comparativo entre essas duas arquiteturas de fluxo de dados (i.e., ETL e ELT), utilizando um conjunto de dados real e ferramentas como Python, PostgreSQL e DBT [7]. Adicionalmente, o contexto histórico e tecnológico da integração de dados e das arquiteturas ETL e ELT será explorado, destacando como elas evoluíram para atender às demandas atuais de engenharia de dados. Ao demonstrar a aplicação prática de ambas as abordagens, este trabalho visa explorar o mundo dos fluxos de dados e contribuir para o debate entre as melhores práticas na construção de fluxos de dados.

1.2 Contextualização

Durante décadas, data warehouses exigiam que os dados fossem transformados antes de sua inserção, devido às limitações das capacidades de processamento dos bancos de dados relacionais para transformações complexas [5]. No entanto, o aumento exponencial do volume de dados gerados nos últimos anos e a ascensão de Big Data trouxe uma nova perspectiva sobre como lidar com informações, exigindo arquiteturas escaláveis e distribuídas para processar, armazenar e analisar dados em larga escala [1].

Com a crescente diversificação das fontes de dados — incluindo sensores IoT, redes sociais e aplicações empresariais — e a necessidade de processamento eficiente, as arquiteturas tradicionais de ETL começaram a se mostrar limitadas. Apesar do ETL ser

vantajoso para cenários que exigem padronização rigorosa antes do armazenamento, sua complexidade operacional e dificuldade em lidar com grandes volumes o tornam menos ideal para o contexto de big data [8]. Nesse cenário, a ascensão de bancos de dados em nuvem [9], com capacidade elástica e arquiteturas otimizadas para processamento distribuído, permitiu a consolidação de ELT, onde os dados são inseridos mais rapidamente e transformados sob demanda [4].

No entanto, a escolha entre ETL e ELT não é uma questão de "um ou outro" [3, 4]. Ambas abordagens de extração de dados ainda desempenham papéis importantes em diferentes contextos e tipos de arquitetura. ELT, com sua maior escalabilidade e capacidade de processamento distribuído, é altamente eficaz quando os dados são volumosos e diversos, típicos de cenários em nuvem. Por outro lado, ETL continua relevante em ambientes onde a transformação à priori é essencial, ou em sistemas que não tem a possibilidade de migrar para tecnologias em nuvem. Além disso, enquanto ELT pode ser mais eficiente em termos de performance no processamento em larga escala, ETL tem uma abordagem mais rigorosa e controlada sobre a qualidade dos dados antes de serem inseridos na base de dados, o que pode ser crucial em algumas indústrias.

O estudo de caso proposto neste trabalho visa justamente explorar essa comparação no contexto específico de uma organização que utiliza tanto soluções em nuvem quanto locais. A análise das escolhas entre as duas abordagens permitirá identificar não apenas qual delas é mais eficiente em termos de custo e desempenho, mas também qual delas melhor atende às necessidades de governança de dados, segurança e escalabilidade.

1.3 Objetivos do estudo de caso

Esta pesquisa tem como objetivo comparar as arquiteturas ETL e ELT no contexto de fluxos de dados. Para isso, foram desenvolvidos dois fluxos: um utilizando a abordagem ETL, com transformação dos dados em Python antes da carga dos dados gerenciados pelo sistema PostgreSQL, e outro seguindo a metodologia ELT, com transformação dos dados diretamente em uma base de dados PostgreSQL usando a ferramenta DBT.

O estudo usa um conjunto de dados público do Kaggle para demonstrar as etapas de limpeza, transformação e carga em cada abordagem de extração de dados considerada. Ao final, será feita uma análise comparativa das duas arquiteturas, destacando suas vantagens, desvantagens e casos de uso ideais. Esta análise será baseada em critérios como escalabilidade, manutenibilidade e adequação aos diferentes cenários criados, contribuindo para a compreensão das melhores práticas na engenharia de dados.

2 Conceitos básicos

2.1 Histórico da Integração de Dados

A integração de dados se tornou uma necessidade crescente à medida que as empresas começaram a lidar com volumes cada vez maiores de informações originadas de fonte diversas. Inicialmente, os dados eram armazenados e processados de forma isolada em sistemas distintos, dificultando a obtenção de uma visão unificada e abrangente das informações. A falta de uma abordagem integrada prejudicava a análise e a tomada de decisão, uma vez que dados de diferentes sistemas não eram facilmente integrados [10].

Nas décadas de 1970 e 1980, os primeiros esforços para integrar dados começaram a surgir, graças ao surgimento dos bancos de dados relacionais. Essas tecnologias permitiram organizar dados em tabelas interligadas, facilitando o acesso e a consulta a grandes volumes de dados. No entanto, as soluções da época eram voltadas principalmente para dados transacionais, limitando sua capacidade de lidar com dados analíticos, que são essenciais para uma análise mais estratégica das informações.

2.1.1 Data Warehouses

O conceito de Data Warehouse (DW) surgiu na década de 1980 como um dos marcos na evolução dos sistemas de processamento de dados. Bill Inmon, considerado o "pai dos Data Warehouses", propôs um modelo de repositório centralizado para integrar e consolidar grandes volumes de dados históricos [2]. Antes disso, os dados eram armazenados separadamente em sistemas operacionais, dificultando a realização de análises que exigiam a combinação de informações de diferentes fontes de dados.

Com o avanço da tecnologia nos anos 1990, a crescente necessidade de ferramentas analíticas impulsionou a adoção dos sistemas de Business Intelligence (BI). Dessa forma, o conceito de DW foi colocado como a solução para o momento, com a proposta de separar operações transacionais dos dados analíticos, criando um repositório exclusivo que permite o processamento de consultas OLAP (On-Line Analytical Processing), mas sem impactar o desempenho dos sistemas operacionais. Esse modelo arquitetural estruturado facilitava a análise minuciosa dos dados e proporcionava visões embasadas em dados reais. As decisões estratégicas, portanto, seriam tomadas com maior clareza e precisão.

Para viabilizar essa integração de dados, tornou-se fundamental a adoção de processos ETL (Extract, Transform, Load), responsáveis por extrair informações de diferentes fontes, transformá-las para um formato adequado e carregá-las em uma base de dados analítica centralizada. Durante essa década, grandes empresas como IBM, Oracle e Microsoft passaram a oferecer soluções robustas de DW, impulsionando sua adoção em larga escala.

2.1.2 Data Lakes

A década de 2000 viu uma explosão vertiginosa no volume de dados, como mostrado na Figura 1. Esse crescimento veio do surgimento de novas fontes de dados, como dados não estruturados, logs e informações provenientes de redes sociais. Mostrou-se, então, a necessidade da evoluir o estado da arte da integração de dados na época.

Figura 1: Aumento da Geração de dados nos anos 2000 [6]

Para lidar com esse aumento exponencial, surgiu o conceito de Data Lake (DL) [11], uma solução que ganhou popularidade no final da década de 2000 como alternativa aos modelos tradicionais de armazenamento de dados estruturado. O termo foi popularizado por Dixon [11], que descreveu o Data Lake como um repositório centralizado para armazenar dados em seu formato bruto, sem a necessidade de estruturação prévia. Diferente de DW, que exigem que os dados sejam transformados antes do armazenamento, os DL oferecem maior flexibilidade ao permitir o armazenamento de qualquer tipo de dado, estruturado ou não. Isso facilitou a inserção de grandes volumes de informações nos bancos de dados, incluindo logs de servidores, imagens, vídeos e interações em redes sociais [12].

Com o avanço das tecnologias de Big Data, como Hadoop e Spark [13], DL se tornaram escaláveis e economicamente viáveis, permitindo que empresas armazenassem e processassem grandes volumes de dados de maneira eficiente. Novas soluções e tecnologias, como Apache Kafka [14] e plataformas de integração em tempo real, também surgiram para atender à crescente demanda por análise instantânea de dados [15].

Apesar das vantagens operacionais, os DL possuem desafios relacionados à governança de dados [12]. Como os dados são armazenados sem transformação, há riscos de inconsistências, duplicações e falta de controle sobre a geração dos dados, o que pode comprometer a confiabilidade das análises. Para lidar com isso, surgiram ferramentas de governança baseadas na catalogação e organização dos dados armazenados, como o AWS Glue Data Catalog [16]. Essas soluções permitem rastrear a geração dos dados, controlar acessos e garantir a qualidade das informações inseridos.

2.1.3 Mudanças nas necessidades empresariais ao longo do tempo

As necessidades empresariais são vivas e se moldam ao mercado, consequentemente acompanhando mudanças nas arquiteturas de armazenamento e processamento de dados. Inicialmente, o foco estava na automação dos processos operacionais e na eficiência das transações, o que levou ao desenvolvimento dos primeiros bancos de dados relacionais. Porém, com o crescimento da extração e carregamento de dados, a inteligência empresarial se tornou uma prioridade, impulsionando o surgimento de DW como solução para centralizar e organizar dados de diversas fontes.

Nos últimos anos, com o avanço de tecnologias como machine learning, inteligência artificial e análise em tempo real, as empresas passaram a exigir maior flexibilidade e capacidade de processamento instantâneo. Isso impulsionou o desenvolvimento de DL, que não apenas permitem armazenar dados em seu formato original, mas também oferecem escalabilidade para atender a novos requisitos analíticos.

Paralelamente, a governança e a segurança dos dados tornaram-se temas centrais e constantemente discutidos, impulsionados pela Lei Geral de Proteção de Dados (LGPD), que destacou a importância da privacidade e conformidade. Com todas essas transformações, a integração de dados deixou de ser apenas uma estratégia técnica e passou a ser uma necessidade vital para as empresas. Como enfatizado em [10], a capacidade de integrar dados de maneira eficaz, aliada a práticas de governança e conformidade, tornou-se um dos principais fatores para o sucesso organizacional na era digital.

2.2 ETL (Extract, Transform, Load)

2.2.1 Origem

O conceito de ETL surgiu na década de 1970 como uma solução para consolidar dados de diferentes fontes em sistemas analíticos. A necessidade dessa abordagem se intensificou com o crescimento de bases de dados empresariais e a DW nos anos 1990. ETL foi projetado para garantir que os dados extraídos de fontes heterogêneas fossem transformados em um formato padronizado antes de serem carregados em um repositório de dados centralizado [2].

2.2.2 Arquitetura

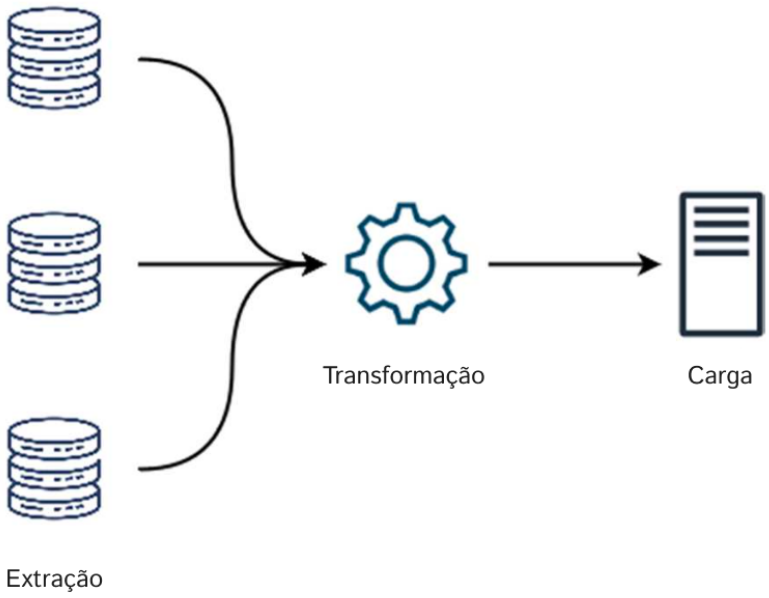

Figura 2: Arquitetura ETL (adaptada de [4])

A arquitetura de um ambiente de ETL é composta por três etapas principais:

- **Extração (Extract)**: Dados são coletados de diversas fontes, como bancos de dados relacionais, arquivos CSV, APIs e sistemas legados.
- **Transformação (Transform)**: Dados passam por processos de limpeza, agregação, padronização e enriquecimento. Essa etapa pode incluir normalização, desnormalização, deduplicação e cálculos específicos [8].
- **Carga (Load)**: Após a transformação, os dados são inseridos no DW, muitas vezes utilizando um modelo multidimensional de dados, para otimizar as análises [2].

A arquitetura ETL tradicional é baseada em processamento em lote, com execuções periódicas dos fluxos de dados. Essa arquitetura permite garantir a qualidade e integridade dos dados antes de sua disponibilização para análise.

Os dados carregados em um DW são frequentemente organizados seguindo um modelo multidimensional, que otimiza consultas analíticas ao estruturar a informação em fatos e dimensões. Nesse modelo, os dados podem ser organizados em esquemas como: 1) o esquema estrela (star schema), no qual uma tabela central de fatos (conhecida como tabela fato) é conectada a tabelas de dimensão, e 2) o esquema floco de neve (snowflake schema), onde as tabelas de dimensão são normalizadas para reduzir redundâncias [8]. Essa abordagem melhora a eficiência das consultas OLAP (On-Line Analytical Processing), facilitando a análise de grandes volumes de dados [2].

2.2.3 Ferramentas de ETL

Desde os primórdios do conceito de ETL, diversas tecnologias têm sido desenvolvidas para facilitar a extração, transformação e carga de dados. Essas tecnologias evoluíram conforme as demandas por processamento de grandes volumes de dados aumentaram. A seguir, são listadas algumas das principais ferramentas e tecnologias utilizadas em processos de ETL, divididas por categorias:

- **Ferramentas de ETL Tradicionais**: Soluções como IBM Information Server, SAS Data Integration Studio e Informatica PowerCenter [17, 18] surgiram nos anos 90 e se consolidaram no mercado corporativo. Elas oferecem ambientes gráficos para o desenvolvimento de fluxos de dados e possibilitam integrações com diversas fontes de dados.

- **Serviço de Integração de Dados**: Introduzido no SQL Server 2005, o Microsoft SQL Server Integration Services (SSIS) [19] oferece um serviço completo para a construção de fluxos ETL dentro do ambiente Microsoft, permitindo integração e transformação de dados em larga escala.

- **Plataforma de Processamento de Dados em Tempo Real**: Criado para lidar com fluxos de dados em tempo real, o Apache Nifi [20] tornou-se uma solução moderna para ETL em ambientes distribuídos, permitindo automação e controle de fluxos de dados em tempo real.

- **Linguagem de Programação**: Com bibliotecas como pandas, numpy, pyodbc, SQLAlchemy e Airflow, Python [21] se tornou uma das linguagens mais populares para automação de tarefas de ETL e transformações de dados, sendo uma opção flexível e extensível para diferentes casos de uso.

2.3 ELT (Extract, Load, Transform)

2.3.1 Origem e Motivação

ELT surgiu como uma abordagem alternativa ao ETL, impulsionado pelo avanço das tecnologias de armazenamento e processamento distribuído. Com o crescimento do Big Data e a popularização dos DL, tornou-se viável inserir dados brutos diretamente em um repositório central e realizar as transformações posteriormente, aproveitando a escalabilidade de bancos de dados em nuvem. Além disso, frameworks distribuídos, como Apache Spark [13], permitem o processamento paralelo de grandes volumes de dados, distribuindo a carga de trabalho entre múltiplos nós de uma rede.

2.3.2 Arquitetura

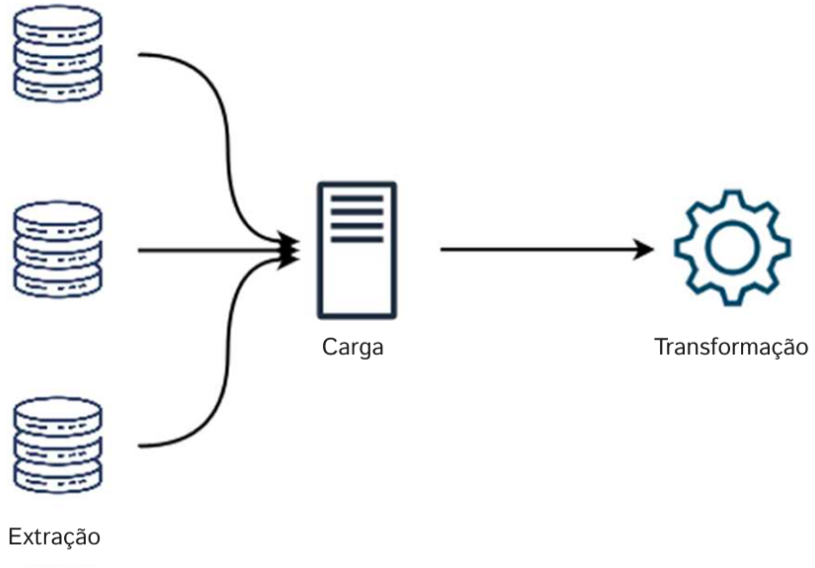

Figura 3: Arquitetura ELT (adaptada de [4])

A arquitetura do ELT segue uma ordem diferente em relação ao ETL:

- **Extração (Extract)**: Dados são coletados de diversas fontes e enviados diretamente para um DL ou DW sem pré-processamento significativo.
- **Carga (Load)**: O dado bruto é inserido no repositório de destino, geralmente sendo armazenado em seu formato original.
- **Transformação (Transform)**: As transformações ocorrem dentro do DW ou DL, utilizando recursos de processamento interno, como Spark, DBT e linguagens de programação para manipulação de dados.

2.3.3 Ferramentas de ELT

Diferentes tecnologias e ferramentas são utilizadas para a implementação de fluxos ELT, dependendo da infraestrutura e dos requisitos do ambiente de processamento de dados. A seguir, são listadas algumas das principais soluções utilizadas:

- **Bancos de Dados em Nuvem**: Soluções como Google BigQuery [22], Amazon Redshift [23] e Snowflake [24] permitem armazenamento escalável e execução de consultas SQL sobre grandes volumes de dados. Essas plataformas são otimizadas para processamento massivo e transformações internas, reduzindo a necessidade de middleware e promovendo a criação de um fluxo ELT eficiente.

- **Framework de Processamento Distribuído**: O Apache Spark é um framework amplamente utilizado para processamento distribuído de grandes volumes de dados, permitindo transformações complexas e escaláveis em pipelines ELT. Ele possibilita a execução de operações diretamente sobre dados armazenados em bancos de dados ou sistemas distribuídos.

- **Linguagem de Programação**: Python tem sido amplamente utilizado em cenários de ELT [25], com bibliotecas como PySpark para processamento distribuído e pandas para manipulação eficiente de dados. Ela tem sido frequentemente empregada [25] em conjuntos com bancos de dados em nuvem e frameworks como Apache Spark para realizar transformações diretamente na camada de armazenamento.

- **Ferramenta de Modelagem e Transformação de Dados**: o DBT é uma das ferramentas mais recentes no cenário de ELT, permitindo a orquestração e transformação de dados diretamente em bancos de dados baseados em SQL. Ele possibilita a criação e manutenção de modelos de dados, facilitando a implementação de transformações no próprio ambiente de armazenamento.

2.4 Conclusão

A evolução da integração de dados reflete diretamente as mudanças tecnológicas e as crescentes necessidades das empresas ao longo das décadas. Desde os primeiros esforços com bancos de dados relacionais até a adoção de W e, posteriormente, DL, observa-se uma busca constante por eficiência, escalabilidade e flexibilidade no armazenamento e processamento de dados.

O avanço das arquiteturas de dados trouxe consigo novas demandas, como governança e segurança. Paralelamente, os processos de ETL desempenharam um papel central ao possibilitar a extração, transformação e carga de dados para repositórios analíticos, permitindo análises mais aprofundadas e permitindo a tomada de decisão baseada em dados.

Com a ascensão de Big Data e dos processos ELT, a integração de dados continua a evoluir, indicando que as arquiteturas tradicionais precisarão se adaptar para atender às exigências da era digital. Assim, compreender os fundamentos da integração de dados, incluindo suas tecnologias e diferentes arquiteturas disponíveis, torna-se essencial para a construção de soluções analíticas eficientes e alinhadas com os desafios do cenário atual.

Este capítulo apresentou a evolução da integração de dados, destacando os avanços das arquiteturas e os impactos das abordagens ETL e ELT. No próximo capítulo, essa discussão será aprofundada com uma análise comparativa dessas duas arquiteturas, considerando suas aplicações práticas e implicações no contexto de pipelines de dados.

3 Estado da Arte

3.1 Comparação entre ELT e ELT na Construção de DW

Haryono et al. [3] realizaram um estudo qualitativo comparando a implementação de DW utilizando os métodos ETL e ELT em diferentes setores industriais. O estudo teve como objetivo identificar as principais vantagens e desafios de cada abordagem a partir da experiência de profissionais da área.

Os resultados indicaram que ELT se destaca pela sua escalabilidade, flexibilidade e suporte a grandes volumes de dados, além da compatibilidade com dados não estruturados. Além disso, apresenta menor custo operacional e maior disponibilidade dos dados, já que a transformação ocorre diretamente no ambiente analítico. Por outro lado, ETL demonstrou benefícios em termos de qualidade dos dados, maior controle sobre as transformações e estabilidade dos pipelines, sendo uma abordagem mais consolidada no mercado. Além disso, a curva de aprendizado de ETL tende a ser mais acessível para profissionais acostumados a práticas tradicionais de integração.

Com base nessas observações, os autores recomendaram ELT para cenários que exigem alto desempenho e flexibilidade na manipulação de dados em larga escala, enquanto ETL é mais adequado para ambientes que demandam um controle rigoroso da qualidade dos dados ou possuem infraestrutura computacional mais limitada.

[3] contribui significativamente para a escolha estratégica entre ETL e ELT, fornecendo uma base comparativa detalhada sobre as vantagens e limitações de cada abordagem. No entanto, ele se limita ao aspecto qualitativo, e não explora de forma prática a implementação dos dois métodos em um mesmo ambiente e com um conjunto padronizado de métricas de avaliação. Para preencher essa lacuna, este trabalho propõe um estudo de caso comparativo, aplicando ETL e ELT sobre um mesmo conjunto de dados, mensurando seu impacto em termos de tempo de processamento, uso de recursos computacionais e qualidade dos dados resultantes.

3.2 Busca por Cenários Ideais de ETL e ELT

Em [26], Dhamotharan Seenivasan realizou uma análise comparativa das arquiteturas ETL e ELT na implementação de DWs. O estudo enfatizou a relevância dessas abordagens na gestão de dados e nas soluções de Business Intelligence (BI), especialmente no contexto do Big Data, onde organizações necessitavam processar e integrar grandes volumes de informações provenientes de diversas fontes.

O autor examinou as duas arquiteturas em termos de desempenho, escalabilidade e custo. Ele destacou que ELT apresentou maior velocidade em cargas massivas, pois postergou as transformações para o pós-carregamento, explorando o poder computacional

dos DWs modernos. Em contrapartida, ETL foi mais lento devido à necessidade de realizar as transformações antes da carga. No quesito escalabilidade, ELT se sobressaiu, sobretudo em ambientes de nuvem, em razão da elasticidade dos recursos computacionais disponíveis. Quanto ao custo, ETL demandou um investimento inicial mais elevado, pois exigiu uma infraestrutura dedicada ao processamento de dados, enquanto ELT incorreu em custos operacionais mais altos, dependendo da frequência e do volume de dados processados diretamente no DW.

Além disso, [26] discutiu cenários ideais para a aplicação de cada abordagem. Para pequenas e médias empresas (PMEs), ETL foi vantajoso quando havia um volume de dados mais controlado e requisitos de transformação bem definidos. Já ELT se destacou em organizações que utilizaram DWs em nuvem e necessitaram de maior escalabilidade. Em grandes corporações, a combinação das duas estratégias foi benéfica, utilizando ETL para transformações críticas antes da carga e ELT para processamentos complexos que exploraram a capacidade computacional do DW. Em infraestruturas locais (on-premises), ETL foi preferível devido ao maior controle sobre os processos de transformação, enquanto, na nuvem, a flexibilidade de ELT proporcionou maior eficiência.

O estudo concluiu que não houve uma solução universalmente ideal, pois a escolha entre ETL e ELT precisou considerar fatores como volume de dados, complexidade das transformações, conformidade regulatória e infraestrutura disponível. ETL foi mais adequado para organizações que priorizaram um controle rigoroso sobre a qualidade e estrutura dos dados antes da carga, especialmente em ambientes legados, onde sistemas antigos não suportavam transformações avançadas dentro do próprio DW. Por outro lado, ELT se destacou no processamento de grandes volumes de dados brutos e em cenários que exigiram escalabilidade e flexibilidade, como análises em tempo real.

No entanto, o estudo de Seenivasan concentrou-se em uma análise conceitual e qualitativa, sem a realização de experimentos práticos. Diferentemente, este trabalho busca validar empiricamente essas premissas por meio de um estudo de caso aplicado. Para isso, avaliou-se o impacto das abordagens ETL e ELT em um ambiente controlado, mensurando seu desempenho, escalabilidade e qualidade dos dados. Ao adotar uma abordagem experimental, esta pesquisa contribui para uma compreensão mais objetiva e baseada em evidências sobre as implicações práticas de cada estratégia, permitindo uma avaliação mais precisa de sua adequação a diferentes cenários organizacionais.

3.3 ETL, ELT e Reverse ETL: Um Estudo de Caso Corporativo

[4] analisaram e compararam três arquiteturas de processamento de dados utilizadas em data warehouses: ETL, ELT e Reverse ETL. O estudo teve como objetivo investigar o impacto dessas abordagens na gestão de dados corporativos, avaliando aspectos como custos, desempenho e adaptação às novas tendências tecnológicas.

Os autores definiram um DW como um repositório que integra dados de múltiplas fontes, estruturados para análise e relatórios. O estudo explorou as duas abordagens tradicionais de processamento de dados, ETL e ELT, além de introduzir o Reverse ETL, que realiza o fluxo inverso dessas arquiteturas, extraindo dados do DW e enviando-os para aplicações operacionais, como CRM (Customer Relationship Management) e ferramentas de marketing e vendas, permitindo um uso mais dinâmico das informações.

A análise evidenciou que a transformação de dados foi a etapa mais complexa tanto no ETL quanto no ELT. ELT apresentou vantagens na análise de grandes volumes de dados, especialmente em DWs modernos, enquanto ETL demonstrou maior adequação a sistemas legados, onde a governança de dados e a padronização são mais críticas. O Reverse ETL, por sua vez, aprimorou a integração entre setores como marketing e vendas, automatizando processos e tornando os dados mais acessíveis para uso operacional.

Os autores concluíram que ETL permaneceu essencial para infraestruturas tradicionais devido à necessidade de maior controle sobre os dados antes do carregamento. Por outro lado, ELT se consolidou como uma abordagem mais alinhada às tecnologias em nuvem, pois se beneficiou da escalabilidade dos DWs modernos. O Reverse ETL se destacou como um complemento estratégico, facilitando a distribuição de dados tratados para sistemas transacionais.

No entanto, o estudo de Singhal e Aggarwal concentrou-se na análise qualitativa de um caso corporativo específico, sem uma comparação quantitativa direta entre as abordagens em um ambiente padronizado. Diferentemente, este trabalho propõe um estudo experimental, avaliando empiricamente os impactos de ETL e ELT em termos de desempenho, consumo de recursos computacionais e qualidade dos dados. Dessa forma, esta pesquisa contribui para a literatura ao fornecer uma análise baseada em métricas concretas, permitindo uma avaliação objetiva das diferenças entre as arquiteturas em um cenário controlado.

3.4 O Processo de Modernização de um Pipeline de Dados

[27] analisou a migração de um pipeline de dados on-premises para uma solução mais escalável e moderna. A abordagem original utilizava ETL com Teradata, empregando stored procedures SQL para transformações. No entanto, essa solução apresentava desafios como a complexidade na gestão de DDL (Data Definition Language), dificuldades na implementação de dados históricos e limitação da escalabilidade devido à infraestrutura local.

Para superar essas limitações, uma nova solução foi criada, adotando ferramentas modernas como Databricks e DBT, permitindo maior modularidade, automação e redução da dependência de um único fornecedor. A transição ocorreu em duas etapas: primeiro, a conversão das stored procedures para DBT, seguida pela migração da execução para a

nuvem com Databricks. O uso do DBT simplificou a manutenção do código e viabilizou uma abordagem bitemporal para tratar dados históricos, ou seja, o armazenamento de informações tanto no tempo de validade quanto no tempo de registro, permitindo consultar os dados como eram em pontos passados. Já a nuvem proporcionou maior escalabilidade e desempenho.

A nova arquitetura seguiu a arquitetura ELT, onde os dados foram extraídos das fontes operacionais e carregados diretamente para a nuvem, com as transformações ocorrendo posteriormente. Como resultado, observou-se uma melhora na confiabilidade dos dados e na automação do pipeline, apesar do aumento na complexidade para o processamento de dados históricos.

Entretanto, [27] teve foco em um caso corporativo específico e não apresentou uma comparação quantitativa padronizada entre as abordagens. Diferentemente, este trabalho propõe uma análise experimental dos impactos de ETL e ELT, mensurando desempenho, escalabilidade e qualidade dos dados. Dessa forma, busca-se oferecer uma avaliação objetiva e baseada em métricas concretas, complementando a literatura existente.

3.5 Conclusão

Embora diversos estudos tenham abordado a comparação entre os métodos ETL e ELT, como os de Haryono [3], Sighal [4] e Seenivasan [26], e também a modernização de pipelines de dados utilizando novas ferramentas como Databricks e DBT, conforme discutido por Pulkka [27], não foram identificados estudos que comparem a implementação prática de ambos os métodos em um mesmo ambiente, utilizando um mesmo conjunto de dados, com foco na extração de dados a partir de fontes abertas, como o Kaggle, e no uso de Python e DBT para construção dos pipelines. Enquanto os artigos citados focam em comparações qualitativas, o trabalho proposto neste documento se diferencia por realizar essa comparação de forma aplicada, construindo dois pipelines distintos—um baseado na arquitetura ETL, onde as transformações ocorrem antes da carga no banco de dados PostgreSQL, e outro seguindo a arquitetura ELT, com transformações realizadas posteriormente no DBT. Dessa forma, este estudo contribui para o avanço no estado da arte ao oferecer uma análise prática e detalhada das diferenças entre ETL e ELT em um cenário controlado, evidenciando suas implicações para desempenho, escalabilidade e manutenibilidade. Por fim, não se tem conhecimento da realização de um estudo de caso que compare ETL e ELT, tornando o trabalho aqui descrito original e inovador.

4 Estudo de Caso: Pipelines de ETL e ELT

Este estudo adota uma abordagem comparativa entre as arquiteturas de integração de dados ETL e ELT, com o objetivo de entender as diferenças e vantagens de cada uma dessas arquiteturas em um cenário de processamento de dados reais. A principal ideia é analisar as etapas de extração, transformação e carregamento dos dados, destacando os desafios e as características específicas de cada abordagem, com o objetivo de compreender como essas arquiteturas se aplicam na prática e suas implicações.

Para isso, dois fluxos foram criados para processar o mesmo conjunto de dados, escolhido por suas características típicas de um conjunto de dados não tratado, como dados desestruturados, inconsistências e informações indesejadas. Esses critérios de limpeza de dados estão definidos na Seção 4.2.1. A partir dessa definição, os fluxos foram implementados, permitindo observar de forma prática como as abordagens de ETL e ELT lidam com esses dados e extrair suas respectivas vantagens e desvantagens.

Após a definição e construção dos fluxos, foi realizada uma análise levando em consideração critérios como escalabilidade, facilidade de manutenção, flexibilidade e a complexidade das transformações necessárias, os quais estão detalhados na Seção 4.1. Além disso, na Seção 4.2.2, é feita uma avaliação das ferramentas escolhidas para cada contexto, explorando como elas impactam a eficiência e a viabilidade de cada abordagem. Na Seção 4.3, é iniciado o estudo de caso prático, com detalhes sobre os pré-requisitos e o desenvolvimento das duas arquiteturas aplicadas. A Seção 4.4 avalia as arquiteturas em termos de escalabilidade, manutenibilidade, além de elencar desafios e vantagens, abordando também os cenários ideais para a aplicação de cada uma das arquiteturas demonstradas. Por fim, é realizado um estudo sobre as tendências futuras dentro do contexto de pipelines de dados.

O estudo de caso pode ser encontrado no Repositório do Github.

4.1 Métricas de Comparação entre ETL e ELT

Para estabelecer uma comparação objetiva entre as arquiteturas ETL e ELT, é necessário definir um conjunto de métricas que permitam avaliar suas características em diferentes cenários [3, 4]. Consideram-se dois contextos distintos: (i) um conjunto de dados reduzido, similar ao utilizado neste estudo, e (ii) um conjunto de dados maior, representando cenários de grande escala. As métricas aqui propostas são:

1. **Tempo de Processamento**: Mede o tempo total necessário para extrair, transformar e carregar os dados, permitindo avaliar a eficiência de cada abordagem.

2. **Uso de Recursos Computacionais**: Analisa o consumo de CPU e memória

durante a execução dos fluxos ETL e ELT, identificando possíveis gargalos de desempenho.

3. **Facilidade de Manutenção**: Considera a complexidade da implementação e manutenção dos pipelines de dados, avaliando aspectos como modularidade, reutilização de código e necessidade de intervenções manuais.

4. **Escalabilidade**: Examina a capacidade da arquitetura de lidar com volumes crescentes de dados sem comprometer significativamente o desempenho.

5. **Confiabilidade e Qualidade dos Dados**: Avalia a integridade e consistência dos dados transformados, incluindo a ocorrência de erros ou perdas de informação.

6. **Flexibilidade e Adaptação**: Mede a facilidade de adaptação a novas fontes de dados ou mudanças na estrutura dos dados existentes.

Neste estudo, apenas as métricas **3, 4, 5 e 6** foram consideradas para análise, visto que a base de dados escolhida possui um volume de dados reduzido por limitações de tempo do estudo proposto. Apesar do tempo de execução do pipeline de dados ser comentado eventualmente, aspectos como tempo de processamento e consumo extensivo de recursos computacionais não puderam ser explorados em profundidade por falta de tempo hábil para considerá-los e por isso, tais aspectos são indicados posteriormente, como trabalhos futuros. Além disso, as métricas aqui listadas são vistas como contribuição de um trabalho pioneiro nesta área e servir de base para futuras pesquisas que utilizem conjuntos de dados maiores, permitindo uma análise mais abrangente das diferenças entre ETL e ELT em cenários onde seja proposta uma perspectiva de dados em grande escala.

4.2 Definição do Estudo de Caso

Para conduzir a comparação entre as arquiteturas ETL e ELT, foi escolhido um conjunto de dados público proveniente do Kaggle, que contém informações detalhadas sobre jogadores da franquia FIFA 21 da EA Sports.

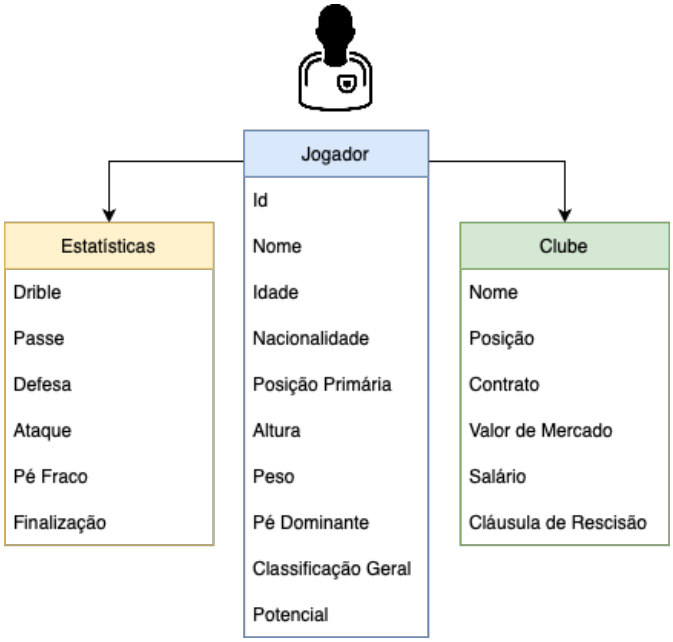

Figura 4: Esquema lógico do conjunto de dados usado

A Figura 4 apresenta o esquema lógico do conjunto de dados, destacando algumas das principais informações presentes, mas sem necessariamente representar sua estrutura completa. Este conjunto de dados foi obtido através de web scraping do Site oficial do SoFIFA, e apresenta diversas características típicas de dados extraídos da web, que frequentemente incluem problemas como formatação inconsistente, presença de valores ausentes, tipos de dados inadequados e a necessidade de padronização de unidades. Esses problemas são comuns em dados não tratados, sendo necessários passos de limpeza e transformação para torná-los utilizáveis para análises futuras. Alguns exemplos de problemas encontrados no referido conjunto de dados incluem:

- **Conversão de colunas não numéricas:** Atributos como **Height** (altura) e **Weight** (peso) estão armazenados como `string` e precisam ser convertidos para valores numéricos, enquanto algumas colunas como **Value**, **Wage** e **Release Clause** contêm caracteres especiais (como **M** para milhões e **€** para euros), que devem ser tratados para facilitar a análise numérica.

- **Remoção de caracteres desnecessários:** Muitas colunas, como **Club** (clube), possuem caracteres de formatação inadequada, como quebras de linha, que devem ser removidos para padronizar as informações.

- **Tratamento de datas e intervalos de tempo:** A coluna **Contract** (contrato) contém formatos variados para o ano de ingresso do jogador no clube, o que requer normalização para permitir cálculos de tempo de permanência.

Esses problemas ilustram as dificuldades comuns encontradas em dados extraídos da web, que frequentemente precisam de um processamento de conversão e limpeza para

garantir que possam ser utilizados de forma eficaz em análises e modelos preditivos. O objetivo deste estudo é processar esse conjunto de dados utilizando as duas arquiteturas — ETL e ELT — para destacar as principais diferenças em termos de limpeza, transformação e carregamento de dados em um banco de dados relacional PostgreSQL. As diferenças serão analisadas em detalhes, considerando como cada método lida com as características dos dados e com os desafios comuns associados à limpeza e transformação de dados.

4.2.1 Limpeza dos Dados

Como comentado na Seção 4.2, o conjunto de dados utilizado contém diversas informações sobre jogadores de futebol, incluindo atributos técnicos, dados contratuais, características físicas e histórico de clubes. Por ter sido extraído via web scraping, os dados apresentam problemas comuns desse tipo de aquisição, como formatação inconsistente, tipagem incorreta e valores ausentes.

Entre os principais desafios identificados no conjunto de dados estão:

- **Conversão de colunas não numéricas:**
  - Explicitado no Quadro 1, atributos como **Height** (altura) e **Weight** (peso) estão armazenados como `string` e precisam ser convertidos para valores numéricos (`int`).

Quadro 1 - Alturas e pesos em diferentes unidades de medida

Height	180cm 171cm 6'5"	189cm 197cm 5'11"	179cm 200cm 6'4"	195cm 166cm 6'1"	172cm 6'2" 6'0"	182cm 164cm 5'10"	186cm 198cm 5'9"	192cm 6'3" 5'6"
Weight	66kg 183lbs 103kg	60kg 179lbs 99kg	94kg 172lbs 102kg	79kg 196lbs 56kg	67kg 176lbs 101kg	65kg 185lbs 57kg	59kg 170lbs 55kg	61kg 203lbs 104kg

  - As colunas **Value** (valor de mercado), **Wage** (salário) e **Release Clause** (cláusula de rescisão), mostradas no Quadro 2, contêm caracteres especiais (**M** para milhões, **K** para milhares, e **€** para euros), e devem ser convertidas para valores numéricos.

Quadro 2 - Valores de mercado, salário e cláusula de rescisão

Value	€103.5M	€63M	€120M	€129M	€132M
Wage	€560K	€220K	€125K	€370K	€270K
Release Clause	€138.4M	€75.9M	€159.4M	€161M	€166.5M

  - A coluna **Hits**, representada pelo Quadro 3, representa a quantidade de chutes a gol do atleta, e contém valores com o sufixo **K** para milhares. Por isso, deve ser convertida para valores numéricos.

Quadro 3 - Quantidade de chutes a gol de um atleta

Hits		
499	4.3K	296
515	943	1.2K
903	335	191

- **Remoção de caracteres desnecessários:**
  - O Quadro 4 mostra a coluna **Club** (clube), que contém quebras de linha desnecessárias. Elas devem ser eliminadas para fins de padronização.

Quadro 4 - Clubes de futebol com formatação incorreta

Club
\n\n\n\nFC Barcelona
\n\n\n\nJuventus
\n\n\n\nAtlético Madrid
\n\n\n\nManchester City
\n\n\n\nParis Saint-Germain

  - As colunas de **habilidades**, exemplificadas pelo Quadro 5, contêm símbolos de estrelas, que devem ser removidos para permitir análise quantitativa.

Quadro 5 - Valores de habilidades

W/F	**S/M**	**I/R**
4 ∗	4 ∗	5 ∗
4 ∗	5 ∗	5 ∗
3 ∗	1 ∗	3 ∗
5 ∗	4 ∗	4 ∗
5 ∗	5 ∗	5 ∗

- **Tratamento de datas:** No Quadro 6, a coluna **Contract** (contrato) contém informações sobre o tipo de contrato, ano de início e fim, todas agregadas em formato textual (string). Essa coluna não será transformada, mas será utilizada para gerar informações valiosas e criar novas colunas, como: o tipo de contrato (se o jogador está livre no mercado, emprestado ou com contrato vigente), o ano de início do contrato e o ano de fim do contrato.

Quadro 6 - Diferentes formatos de contrato dos jogadores

Contract
Jun 30, 2021 On Loan
Free
2019 ~ 2026

O conjunto de dados escolhido requer diversas transformações, incluindo conversão de tipos, remoção de caracteres indesejados e padronização de unidades. Essas necessidades permitem uma análise comparativa entre as arquiteturas ETL e ELT e suas respectivas ferramentas, destacando como cada uma opera em termos de escalabilidade, manutenibilidade, flexibilidade e eficiência no processamento de dados desestruturados.

Esses aspectos serão mensurados seguindo os seguintes critérios:

- **Flexibilidade:** a facilidade de adaptação a novas fontes de dados. Por exemplo, quando há mudanças na estrutura do conjunto de dados, pode ser necessário ajustar o processo para garantir a compatibilidade sem comprometer a inserção e transformação;
- **Manutenibilidade:** a complexidade de manutenção do sistema. Se novos atributos forem adicionados, a forma como o pipeline lida com essas mudanças pode impactar a necessidade de ajustes na transformação ou no armazenamento;
- **Escalabilidade:** a capacidade de lidar com grandes volumes de dados. O aumento da quantidade de registros pode influenciar o desempenho e a viabilidade do processamento, dependendo da distribuição da carga de trabalho entre extração, transformação e carga;
- **Eficiência:** a rapidez e o uso otimizado dos recursos no processamento dos dados. A forma como certos cálculos e padronizações são aplicados pode afetar o tempo de processamento e o consumo de recursos computacionais.

4.2.2 Escolha de Ferramentas

A seleção das ferramentas foi realizada com base em sua popularidade, robustez e adequação técnica para cada abordagem do estudo (ETL e ELT). Essas ferramentas foram escolhidas por sua capacidade comprovada de otimizar o processamento de dados, além de serem amplamente reconhecidas por suas comunidades e a literatura da área.

- **Python**: A linguagem Python foi escolhida devido à sua ampla adoção na área de análise de dados e automação de processos, o que é reconhecido por sua versatilidade e robustez em diversos contextos. Segundo [25], Python é uma das linguagens mais utilizadas por cientistas de dados e engenheiros de dados, graças ao seu rico ecossistema de bibliotecas como `Pandas`, `NumPy` e outras, que são essenciais para manipulação, análise e visualização de dados.

  No estudo de caso, Python desempenhou um papel fundamental em três etapas principais do processo:

  - **Extração de dados**: As bibliotecas `requests` e `kaggle` foram escolhidas devido à sua eficiência na interação com APIs e no download do conjunto de dados, uma vez que oferecem uma integração direta com a plataforma Kaggle e com outras fontes externas de dados.
  - **Tratamento de dados no fluxo ETL**: As bibliotecas `Pandas` e `NumPy` foram escolhidas por sua facilidade de uso e capacidade de transformação de dados em grande escala.
  - **Inserção de dados**: A biblioteca `SQLAlchemy` foi escolhida pela sua popularidade e pela capacidade de abstrair a comunicação com diferentes bancos de dados relacionais, o que facilita a integração com o PostgreSQL.

- **PostgreSQL**: O PostgreSQL foi escolhido como sistema de banco de dados relacional devido à sua robustez, escalabilidade e compatibilidade com operações de transformação de dados, além de possuir integração com o DBT. De acordo com [28], o PostgreSQL é um dos bancos de dados relacionais mais populares e amplamente adotados, devido à sua capacidade de lidar com grandes volumes de dados.

  No contexto deste estudo, o PostgreSQL foi utilizado como repositório central tanto nos fluxos ETL quanto ELT, proporcionando uma plataforma sólida e de alto desempenho para o armazenamento e processamento de dados. Especificamente no fluxo ELT, DBT aproveitou a capacidade computacional do PostgreSQL para realizar transformações diretamente no banco, tirando proveito de sua arquitetura otimizada para processar grandes volumes de dados.

- **DBT (Data Build Tool)**: DBT é uma ferramenta relativamente recente, criada em 2016 pela DBT Labs. Ela foi escolhida principalmente por sua popularidade crescente e sua abordagem moderna no processamento de dados dentro de pipelines ELT. Como destacado por [29], DBT tem se tornado uma das ferramentas mais populares para transformação de dados devido à sua facilidade de uso, automação de testes de qualidade e pela capacidade de criar transformações reutilizáveis por meio de macros, otimizando a manutenção e a escalabilidade dos processos.

  O DBT foi utilizado em nosso estudo para:

  - **Realizar transformações diretamente no PostgreSQL**, aproveitando sua alta capacidade de processamento e potencializando o poder computacional das transformações de dados.
  - **Utilizar macros para padronização de transformações**, facilitando a reutilização de código e garantindo maior flexibilidade nas transformações de dados. As macros no DBT funcionam como funções parametrizáveis escritas em Jinja e SQL, permitindo a criação de operações reutilizáveis

- **Automatizar testes de qualidade de dados**, garantindo a confiabilidade e consistência das informações.
- **Gerar documentação automaticamente**, o que facilita o entendimento da estrutura e operações realizadas no banco de dados.

4.3 Implementação do Estudo de Caso

4.3.1 Pré-Requisitos

Para garantir que a obtenção dos dados utilizados nesta pesquisa seguisse boas práticas de automação em fluxos de dados, foi realizada a extração diretamente do Kaggle, uma plataforma online que disponibiliza diversos conjuntos de dados prontos para análise. No entanto, antes de iniciar esse processo, as configurações a seguir foram necessárias:

- **Configuração do Python**: Foi necessário garantir a instalação do Python no ambiente de desenvolvimento. A versão mais recente pode ser obtida no site oficial: https://www.python.org.
- **Instalação da biblioteca do Kaggle**: Para obter o conjunto de dados de forma automatizada, instalou-se a biblioteca `kaggle`, que permite interagir com a API da plataforma. A instalação foi realizada com o comando explicitado no Código 1.

```
1    pip install kaggle
```

Código 1: Instalação da biblioteca do Kaggle

- **Configuração da API do Kaggle**: O acesso programático ao Kaggle requer uma chave de autenticação (API Key). Essa chave pode ser gerada na conta do usuário e utilizada para permitir o download automatizado de dados. As instruções detalhadas estão disponíveis na documentação oficial: https://www.kaggle.com/docs/api.
- **Configuração do PostgreSQL**: O PostgreSQL pode ser obtido e configurado através do site oficial: https://www.postgresql.org.
- **Instalação e configuração do Jupyter Notebook**: O Jupyter Notebook foi empregado como ferramenta interativa para desenvolver e documentar a análise dos dados. Ele permite a execução incremental de código Python, além de possibilitar a inclusão de textos explicativos e visualizações de dados no mesmo ambiente. A instalação foi realizada executando o Código 2 no terminal:

```
pip install notebook
```

Código 2: Instalação do Jupyter Notebook

Já o ambiente de desenvolvimento pode ser inicializado pelo Código 3. Esse comando abre uma interface interativa no navegador, permitindo a escrita e execução de código de maneira intuitiva.

```
jupyter notebook
```

Código 3: Comando para iniciar o Jupyter Notebook

- **Instalação e configuração do DBT**: A instalação do DBT foi realizada pela linha de comando do Código 4

```
pip install DBT-core
```

Código 4: Instalação da biblioteca DBT-core

Para configuração detalhada e integração com o banco de dados, as instruções podem ser consultadas na documentação oficial: https://docs.getDBT.com.

4.3.2 Fluxo ETL

Na Figura 5, mostra-se o fluxo ETL escolhido, que consiste em quatro etapas principais: extração, limpeza e tratamento, carga e disponibilização dos dados para consultas.

1. **Extração**: O processo inicia-se com a obtenção do conjunto de dados não tratado do Kaggle. Esses dados brutos são a base inicial para o trabalho e, por serem provenientes de um web scrapping, possuem inconsistências que precisam de atenção.

2. **Limpeza e Tratamento**: Após a extração, os dados passam por uma fase de limpeza e tratamento, onde são removidas duplicatas, corrigidos erros, tratados valores faltantes e normalizados os dados para garantir consistência e qualidade. A normalização pode envolver a padronização de formatos de datas, escalas numéricas ou categorias, preparando os dados para análises futuras.

3. **Carga**: Uma vez tratados, os dados são carregados no banco de dados PostgreSQL, permitindo o armazenamento estruturado dos dados, facilitando a execução de consultas complexas e a manipulação eficiente das informações.

4. **Disponibilização para Consultas**: Com os dados devidamente tratados e armazenados, eles estão prontos para serem consultados e analisados. A partir daí, possível realizar consultas SQL, gerar visualizações, aplicar técnicas de análise de

dados ou modelos de aprendizagem de máquina, dependendo dos objetivos do trabalho.

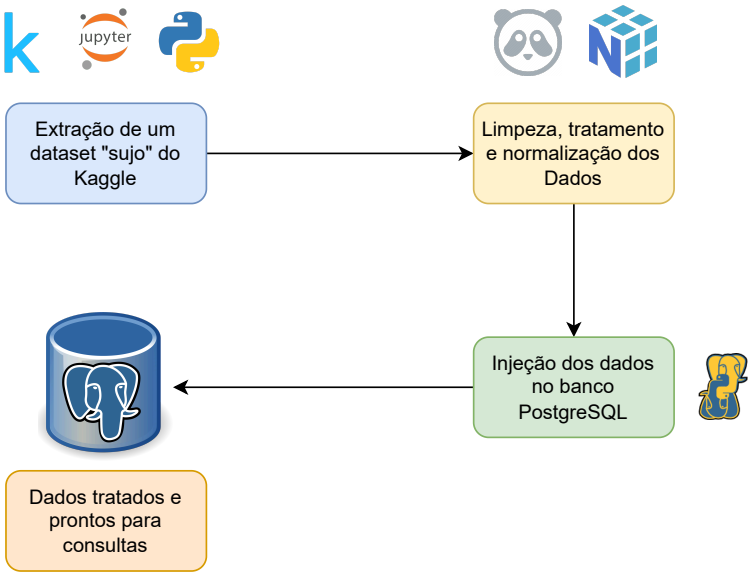

Figura 5: Fluxo ETL do estudo de caso

O processo de download dos dados é realizado automaticamente através do Código 5 em Python, onde é criada uma pasta chamada `dataset`, onde os arquivos baixados são armazenados. Em seguida, é usado um comando que solicita ao Kaggle, o download do conjunto de dados desejado. Quando a operação é concluída, é exibida uma mensagem informando que o arquivo foi salvo com sucesso.

```
conjunto de dados_dir = "conjuntos de dados"
os.makedirs(conjunto de dados_dir, exist_ok=True)
kaggle_conjunto de dados = "yagunnersya/fifa-21-messy-raw-
    conjunto de dados-for-cleaning-exploring"
os.system(f"kaggle conjuntos de dados download -d {
    kaggle_conjunto de dados} -p {conjunto de dados_dir}")
print(f"conjunto de dados baixado e salvo em: {conjunto de
    dados_dir}")
```

Código 5: Download do conjunto de dados do Kaggle

Com o conjunto de dados extraído e inserido em um *dataframe*, inicia-se a etapa

de *Limpeza dos Dados*. Para lidar com as problemáticas do conjunto de dados apontadas na Seção 4.2.1, foram realizados os processos a seguir:

- **Padronizações**: A padronização de nomes de colunas, considerada uma boa prática em banco de dados, facilita a manipulação e consistência dos dados, evita erros em consultas, favorece a integração de dados e garante maior clareza e organização no banco de dados.

  O Quadro 7 apresenta colunas não padronizadas do conjunto de dados:

  Quadro 7 - Colunas não Padronizadas

Name	**↓OVA**	**Preferred Foot**
Best Position	**Loan Date End**	**Release Clause**
Heading Accuracy	**GK Reflexes**	**W/F**

  A padronização segue algumas diretrizes, como:

  - **Sem espaços**: Todos os espaços devem ser substituídos por underscores (_), evitando problemas de sintaxe e facilitando a manipulação dos dados em diferentes sistemas e consultas. Por exemplo, `"Loan Date End"` deve se tornar `"Loan_Date_End"`.
  - **Sem caracteres especiais**: É fundamental evitar caracteres especiais, como acentos, hífens e símbolos que podem causar problemas em consultas ou scripts. Assim, `"↓OVA"` torna-se `"ova"` e `"W/F"` torna-se `"wf"`.
  - **Padronização de letras**: Para garantir uniformidade, as colunas devem ser consistentes em relação ao tipo de letra que utilizam, podendo ser maiúsculas ou minúsculas. Isso elimina a confusão gerada por diferentes convenções de capitalização e facilita a leitura dos nomes das colunas. Nesse estudo de caso, as colunas foram padronizadas para letras minúsculas. Por exemplo, `"GK Reflexes"` se torna `"gk_reflexes"`.

  No estudo de caso, foi aplicada a padronização e normalização de colunas. No Código 6, algumas dessas práticas foram aplicadas, como explicado a seguir:

  - Em `str.lower().str.replace(' ', '_').str.replace('/', '')`, os nomes das colunas foram colocados em letra minúscula, depois os espaços foram substituídos por (_) e, por fim, (/) foram retirados.
  - A coluna `"↓OVA"` foi renomeada para `"ova"`, retirando assim o caractere especial (↓).

```
df.columns =
  df.columns.str.lower().str.replace(' ', '_').str.replace('/', '
   ')

df = df.rename(columns={"↓ova" : "ova"})
```

Código 6: Padronização de Colunas

O resultado final da normalização é destacado no Quadro 8:

Quadro 8 - Colunas Padronizadas

name	**ova**	**preferred_foot**
best_position	**loan_date_end**	**release_clause**
heading_accuracy	**gk_reflexes**	**wf**

- **Conversão de colunas não numéricas**: Foi criada uma função chamada `ft_to_cm`. Essa função tem como objetivo converter uma medida de altura, que pode estar em pés (`ft`) e polegadas (`in`), para centímetros (`cm`). A função funciona da seguinte maneira:

  1. **Verificação de Formato**: A função começa verificando se a *string* fornecida contém um apóstrofo (`'`). Esse caractere é usado para indicar medidas no formato "pés e polegadas"(por exemplo, `5'10"`).
  2. **Separação dos Valores**: Se a *string* contiver um apóstrofo, ela é dividida em duas partes usando `split("'")`. A primeira parte representa os pés e a segunda representa as polegadas.
  3. **Conversão de Valores**:
     - O valor correspondente aos pés é convertido para um número inteiro.
     - O valor das polegadas também é convertido para um número inteiro, removendo o símbolo de polegadas (") se estiver presente. Caso não haja valor para polegadas (por exemplo: `5'`), assume-se que o valor seja zero.
  4. **Cálculo da Conversão**: A conversão é feita aplicando as seguintes regras:
     - Cada pé equivale a `30.48` cm.
     - Cada polegada equivale a `2.54` cm.

     Dessa forma, a conversão para centímetros é dada pela computação do Algoritmo 1.
  5. **Tratamento de Valores em cm**: Se a *string* original não contiver um apóstrofo, a função assume que o valor já está em centímetros. Assim, ela apenas remove a string "cm"e converte o valor para inteiro.

6. **Retorno do Resultado**: A função retorna a altura convertida para centímetros, garantindo um formato numérico consistente para todas as entradas.

Aqui está um exemplo de como a função `ft_to_cm` funciona:

- Se a entrada for `5'10"`, a função retornará `178` (pois 5 pés e 10 polegadas equivalem a aproximadamente 178 centímetros).
- Se a entrada for `180`, a função retornará `180` (assumindo que o valor já está em centímetros).

Algorithm 1 Função Python para Conversão de Pés em Centímetros

```
function FT_TO_CM(x)
    if ‘’‘ em x then
        Separe feet e inches
        Retorne feet × 30.48 + inches × 2.54 como inteiro
    else
        Retorne x sem a string ‘cm‘ e transformado em inteiro
    end if
end function
```

Para tratar valores acerca do peso dos atletas, foi criada uma função chamada `lbs_to_kg`, que converte uma medida de peso, que pode estar em libras (`lbs`), para quilogramas (`kg`):

1. **Verificação de Formato**: A função começa verificando se a *string* fornecida contém a unidade `lbs`, que indica que o valor está em libras (por exemplo, `150 lbs`).
2. **Remoção da Unidade**: Se a unidade `lbs` estiver presente, a função remove a parte textual `lbs` da *string*, deixando apenas o valor numérico (por exemplo, `150 lbs` se torna `150`).
3. **Cálculo da Conversão**: A função então converte o valor de libras para quilogramas. Para isso, divide-se o valor obtido por `2.2` (já que 1lb equivale a aproximadamente 0,4536kg, e `1 / 2.2` é uma aproximação comum) e arredonda o resultado para o número inteiro mais próximo.
4. **Retorno do Resultado**: Se a *string* original não contiver a unidade `lbs`, a função assume que o valor já está em quilogramas, retira a parte textual "kg"e simplesmente retorna o valor inteiro correspondente. Por fim, a função é aplicada à coluna `weight`.

Aqui está um exemplo de como a função `lbs_to_kg`, implementada pelo Algoritmo 2, funciona:

- Se a entrada for `150lbs`, a função retornará `68` (pois 150lbs equivalem a aproximadamente 68kg).
- Se a entrada for `70`, a função retornará `70` (assumindo que o valor já está em kg).

Algorithm 2 Função Python para Conversão de Libras em Quilos

```
1: function LBS_TO_KG(x)
2:     if 'lbs ' em x then
3:         Remova 'lbs ' de x
4:         Calcule kilograms como x dividido por 2.2 e arredondado
5:         Retorne kilograms
6:     else
7:         Remova 'kg' de x
8:         Retorne x como inteiro
9:     end if
10: end function
```

Para lidar com salários, cláusulas de contrato e valor de compra dos jogadores, foi criada uma função chamada `money`, que tem como objetivo converter valores monetários que podem estar expressos em milhões (`M`) ou milhares (`K`) para valores inteiros, removendo também o símbolo da moeda (€). A função funciona da seguinte maneira:

1. **Remoção do Símbolo de Euro**: A função começa verificando se a *string* fornecida contém o símbolo €. Se presente, o símbolo é removido para facilitar o processamento do valor.
2. **Verificação de Milhões (`M`)**: A função verifica se a *string* contém a letra `M`, que indica que o valor está em milhões. Se presente, a função remove o `M` e multiplica o valor numérico por `1.000.000` para converter o valor para sua representação inteira.
3. **Verificação de Milhares (`K`)**: Se a *string* não contiver `M`, mas contiver a letra `K`, que indica que o valor está em milhares, a função remove o `K` e multiplica o valor numérico por `1.000` para convertê-lo para sua representação inteira.
4. **Retorno do Resultado**: Se a *string* não contiver `M` ou `K`, a função assume que o valor já está em sua forma numérica e simplesmente retorna o valor inteiro correspondente.

Após a definição da função `money`, ela é aplicada a três colunas diferentes:

- **value**: A coluna `value` é convertida para valores inteiros e depois dividida por `1.000.000` para representar os valores em milhões.

- **wage**: A coluna `wage` é convertida para valores inteiros, mantendo os valores em sua forma original.
- **release_clause**: A coluna `release_clause` é convertida para valores inteiros e depois dividida por `1.000.000` para representar os valores em milhões.

Em seguida, as colunas são renomeadas para refletir o significado dos valores processados:

- **value**: Renomeada para `values_in_euro_million`, indicando que os valores estão em milhões de euros.
- **wage**: Renomeada para `wage_in_euros`, indicando que os valores representam salários em euros.
- **release_clause**: Renomeada para `release_clause_in_euro_million`, indicando que os valores representam cláusulas de rescisão em milhões de euros.

Aqui está um exemplo de como a função `money`, exemplificada pelo Algoritmo 3, funciona:

- Se a entrada de `values_in_euro_million` for `€5M`, o retorno será `5.0` (representando 5 milhões de euros).
- Se a entrada de `wage_in_euros` for `€150K`, o retorno será `150.000` (representando 150 mil euros).
- Se a entrada de `release_clause_in_euro_million` for `€25M`, o retorno será `25.0` (representando 25 milhões de euros).

Algorithm 3 Função Python para Conversão de Valores Monetários

```
1:  function MONEY(x)
2:      if '€' em x then
3:          Remova '€' de x
4:      end if
5:      if 'M' em x then
6:          Remova 'M' de x
7:          Retorne x × 1.000.000
8:      end if
9:      if 'K' em x then
10:         Remova 'K' de x
11:         Retorne x × 1.000
12:     end if
13:     Retorne x como inteiro
14: end function
```

Outra coluna que passa por uma transformação semelhante é a coluna *Hits*, mostrada no Quadro 3. Para lidar com ela, a função `convert_hits`, explicada pelo Algoritmo 4, foi criada.

Algorithm 4 Função python para Conversão de Chutes à Gol

```
1: function CONVERT_HITS(x)
2:     if x é nada then
3:         Retorne nada
4:     end if
5:     if 'K' em x then
6:         Retorne o valor de x sem o 'K' como float
7:     end if
8:     Retorne x como float
9: end function
```

- **Remoção de caracteres desnecessários**: Para remoção de caracteres que dificultam possíveis cálculos e visualizações, foi criada uma função chamada `remove_star_from_columns`, que tem como objetivo limpar os dados removendo o caractere `*` (estrela) das colunas *wf, sm e ir* do DataFrame.

  A função recebe dois parâmetros:

  - `df`: o DataFrame em questão.
  - `columns`: uma lista com os nomes das colunas nas quais a remoção deve ser aplicada.

  O funcionamento da função `remove_star_from_columns` é implementada pelo Algoritmo 5 e descrita a seguir:

  1. Para cada coluna listada em `columns`, a função substitui todas as ocorrências do caractere `"*"` por um espaço vazio.
  2. O método `.str.replace("*", " ", regex=False)` garante que a substituição seja feita apenas em colunas que contêm texto.
  3. Por fim, a função retorna o DataFrame já modificado.
  4. Em `df = remove_star_from_columns(df, ['wf','sm','ir'])`, a função é aplicada para as colunas `wf, sm e ir`.

  Já a remoção dos `\n\n\n` remanescentes do web scrapping, feito no `Código 7`, é mais simples:

  - O método `.str.strip()` é utilizado para remover caracteres específicos do início e do final dos valores presentes na coluna `"club"`.

Algorithm 5 Função Python para Remoção de Estrelas

```
function REMOVE_STAR_FROM_COLUMNS(df, columns)
    for cada coluna em columns do
        Remova '*' de todos os valores na coluna
    end for
    Retorne df
end function
```

- No caso específico, os caracteres `"\r"` e `"\n"` representam quebras de linha (`carriage return` e `newline`, respectivamente), que podem ter sido inseridas indevidamente durante a extração dos dados.
- Ao aplicar essa função, garante-se que o nome do clube armazenado na coluna `"club"` não contenha espaços ou quebras de linha indesejadas antes ou depois do texto principal.

```
df["Club"]=df["Club"].str.strip("\r\n\r\n\r\n\r\n")
```

Código 7: Remoção de carriage returns e newlines na coluna Club

- **Tratamento de Datas**: Para extrair as informações sobre o contrato do jogador, três funções foram criadas:

1. A função `contract_type`, implementada pelo Algoritmo 6, define o tipo de contrato do jogador com base no conteúdo da string:

Algorithm 6 Função Python para Determinação do Tipo de Contrato

```
function CONTRACT_TYPE(x)
    if "Free"está em x then
        Retorne "Free"
    else if "Loan"está em x then
        Retorne "Loan"
    else if " "está em x then
        Retorne "Contract"
    else
        Retorne pd.NA
    end if
end function
```

2. A função `start_time_contract` determina o ano de início do contrato com base na sua estrutura. Seu funcionamento pode ser visto no Algoritmo 7:
   - A coluna `type_of_contract` é criada aplicando a função `type` à coluna `"contract"`.

Algorithm 7 Função Python para Determinação do Início do Contrato

```
function START_TIME_CONTRACT(x)
    if " "está em x then
        Retorne os primeiros 4 caracteres de x como inteiro
    else if "Loan"está em x then
        Remova "On Loan"de x
        Converta x para data no formato "
        Retorne a data
    else
        Retorne "No Club"
    end if
end function
```

- A coluna `start_year` é criada aplicando a função `start_time_contract` à coluna `"contract"`.
- Valores ausentes em `start_year` e `end_year` são preenchidos com `"No Club"`.

3. A função `end_time_contract` determina o ano de término do contrato com base no tipo de contrato, e é descrita no Algoritmo 8:

Algorithm 8 Função Python para Determinação do Fim do Contrato

```
function END_TIME_CONTRACT(type, contract, loan)
    if type é "Contract" then
        Retorne os últimos 4 caracteres de contract
    else if type é "Loan" then
        Converta loan para data no formato "
        Retorne a data
    else
        Retorne "No Club"
    end if
end function
```

A coluna `end_year` é criada no DataFrame aplicando a função `end_time_contract` às colunas `"type_of_contract"`, `"contract"` e `"loan_date_end"`. Para exemplificar, são mostrados no `Quadro 9`, alguns exemplos de como ficam as colunas `"type_of_contract"`, `"start_year"` e `"end_year"`

Quadro 9 - Exemplos distintos de contratos

type_of_contract	textbfstart_year	**end_year**
Contract	2004	2021
Loan	2021-06-30	2021-06-30
Free	No Club	No Club

- **Inserção dos dados no banco PostgreSQL**: Após o tratamento e a transformação dos dados, o próximo passo foi a inserção desses dados no sistema de banco de dados PostgreSQL, como mostrado pelo `Código 8`. Para isso, foi realizada a conexão com o banco utilizando as credenciais de acesso armazenadas em um arquivo de configuração no formato JSON. Esse arquivo contém informações como o host, a porta, o nome do banco de dados, o usuário e a senha necessários para a conexão.

  A partir dessa configuração, a conexão foi estabelecida utilizando a biblioteca `psycopg2`, que permite a interação com o PostgreSQL, e o `SQLAlchemy`, que facilitou a manipulação e execução de operações no banco. A inserção dos dados no banco foi realizada por meio da função `to_sql()` da biblioteca `pandas`, que permite transferir diretamente um DataFrame para uma tabela no banco de dados. O parâmetro `if_exists='replace'` foi utilizado para garantir que, caso a tabela já existisse no banco, ela fosse substituída pelos novos dados. Isso evita conflitos e garante que a tabela sempre contenha as informações mais atualizadas.

  Em relação ao parâmetro `schema`, a escolha do schema no banco de dados é crucial para a organização das tabelas e dos dados. No processo, foram definidos dois schemas distintos para o armazenamento dos dados:

  - **Dados Não Tratados (Staging)**: dados sem modificações, provenientes de sistemas de origem ou arquivos externos, armazenados em um esquema específico para essa fase. Esse schema serve como área de preparação para dados brutos antes da transformação.
  - **Dados Tratados (Analytics)**: dados limpos e transformados, prontos para análise e relatórios, armazenados em um esquema destinado a abrigar informações analisadas, facilitando consultas e visualizações para análise de desempenho e geração de relatórios.

```
1  with open('config.json', 'r') as config_file:
2      config = json.load(config_file)
3
4  connection = psycopg2.connect(
5      host=config['host'],
6      port=config['port'],
7      database=config['database'],
8      user=config['user'],
9      password=config['password'])
10
11 engine = create_engine(f'postgresql+psycopg2://{config["user"]}:{
       config["password"]}@{config["host"]}:{config["port"]}/{config["
```

```
    database"]}', creator=lambda: connection)

df.to_sql('ETL_clean_fifa_model', engine, schema='analytics',
    if_exists='replace', index=False)
print("DataFrame injetado com sucesso!")
```

Código 8: Inserção dos dados no banco PostgreSQL

Essa etapa finalizou o processo de carga de dados do fluxo ETL, tornando os dados disponíveis para consultas e análises no banco PostgreSQL, com um simples `SELECT * FROM analytics.ETL_clean_fifa_model`, cujo resultado parcial é mostrado no Quadro 10.

Quadro 10 - Amostra dos dados limpos e prontos para consulta

name	**club**	**type_of_contract**	**start_year**	**end_year**
L. Messi	FC Barcelona	Contract	2004	2021
Cristiano Ronaldo	Juventus	Contract	2018	2022
Neymar Jr	Paris Saint-Germain	Contract	2017	2022
G. Bale	Tottenham Hotspur	Loan	2021-06-30	2021-06-30
R. Sigurjónsson	No Club	No Club	No Club	No Club

4.3.3 Fluxo ELT

Na Figura 6, mostra-se o fluxo ELT escolhido, que consiste em três etapas principais: extração, carga e disponibilização no banco de dados, e limpeza, tratamento e disponibilização dos dados limpos no próprio banco.

1. **Extração**: O processo de extração no fluxo ELT é o mesmo apresentado no Código 5, semelhante ao fluxo ETL.

2. **Carga**: Assim como o processo de Extração, o processo de carga no banco de dados *PostgreSQL* é o mesmo apresentado no fluxo ETL, apenas com o conjunto de dados não tratado sendo inserido no esquema *staging*.

3. **Limpeza, Tratamento e Disponibilização para Consultas**: Nessa etapa, mostra-se a maior diferença entre os fluxos apresentados: enquanto o fluxo ETL usa Python, criando funções e aplicando-as nas respectivas colunas, o fluxo ELT usa DBT *(data-built tool)*, uma ferramenta de transformação de dados focada em fluxos ELT, baseada na linguagem **SQL**, comumente usada em bancos de dados.

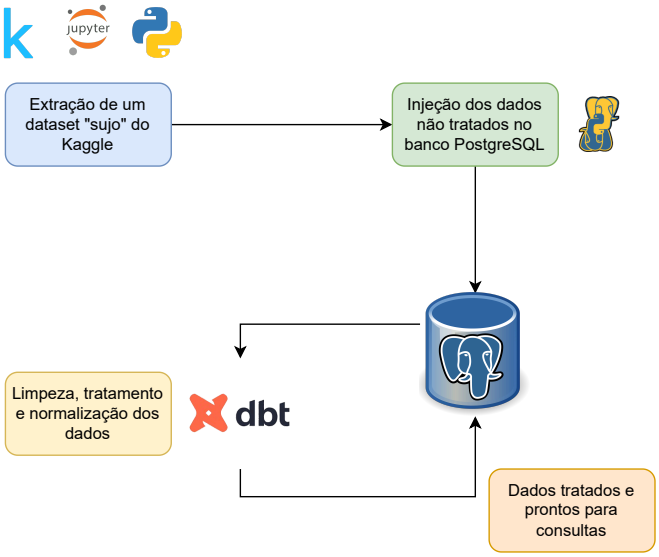

Figura 6: Fluxo ELT no estudo de caso

O DBT permite transformações de dados via **macros**, que são funções reutilizáveis escritas em SQL ou Jinja [30] (uma linguagem de modelagem de templates). Essas macros possibilitam a criação de lógicas complexas e reutilizáveis, como cálculos de métricas e transformações de colunas. Dessa forma, foi possível replicar, no fluxo ELT, os mesmos tratamentos aplicados anteriormente em Python, como a limpeza de dados, a padronização de formatos e a conversão de tipos. Além disso, ao centralizar o processo de transformação, consulta, análise e visualização de dados na mesma linguagem de programação, garantiu-se maior consistência e manutenção do código.

Um exemplo de macro construída é o *strip_affix*, listada no Código 9 e com aplicação exemplificada no Quadro 11, criada com o objetivo de generalizar a extração de prefixos e sufixos, funcionando da seguinte forma:

1. Verifica se o valor na coluna não é nulo;
2. Usa `regexp_replace` para remover o `affix`:
   - `^{{ affix }}` ⇒ Remove o `affix` do início do valor (prefixo).
   - `{{ affix }}$` ⇒ Remove o `affix` do final do valor (sufixo).

3. Se o valor for `NULL`, retorna `NULL`.

```
{% macro strip_affix(column, affix) %}
    CASE
        WHEN {{ column }} IS NOT NULL THEN
            regexp_replace({{ column }}, '^{{ affix }}|{{ affix }}$', '')
        ELSE NULL
    END
{% endmacro %}
```

Código 9: Macro `strip_affix`, que retira prefixos e sufixos dados

Quadro 11 - Funcionamento do macro `strip_affix`

Entrada	affix	Saída
"cm120cm"	"cm"	"120"
"50kg"	"kg"	"50"
"100lbs"	"lbs"	"100"
"25"	"x"	"25"
NULL	"cm"	NULL

A partir daí, foram criadas **macros** que proporcionaram os mesmos resultados gerados pelas funções em *Python*, no fluxo ETL.

Para a transformação dos valores de altura feita na coluna **height**, a `macro ft_to_cm` foi criada e sua lógica ilustrada no Algoritmo 9, reaproveitando o `macro strip_affix`:

Algorithm 9 Macro para Conversão de Alturas de Pés em Centímetros

function FT_TO_CM(coluna)
  if coluna contém "'" **then**
    Remova o sufixo "cm"da coluna usando **strip_affix**
    Separe a parte antes de "'"como pés
    Converta pés para centímetros: pés × 30.48
    Separe a parte depois de "'"como polegadas
    Remova o sufixo '"' de polegadas usando **strip_affix**
    Converta polegadas para centímetros: polegadas × 2.54
    Retorne a soma das conversões como inteiro
  else
    Retorne coluna como inteiro
  end if
end function

Da mesma forma, foi criada a `macro lbs_to_kg` para transformar a coluna **weight**, transformada previamente pela `def lbs_to_kg` no fluxo ETL. Seu funcionamento pode ser entendido pelo Algoritmo 10:

Algorithm 10 Macro para Conversão de Peso de Libras em Quilogramas

```
function LBS_TO_KG(coluna)
    if coluna contém "lbs" then
        Remova o sufixo "lbs"da coluna usando strip_affix
        Converta o valor restante para número
        Converta de libras para quilogramas: valor ÷ 2.2
        Retorne o valor convertido como inteiro
    else
        Retorne coluna como inteiro
    end if
end function
```

Em relação à conversão de valores monetários, transformados no fluxo ETL pela `def money`, a `macro format_money` criada já contempla tanto a retirada dos símbolos `€`, `M` e `K` como a conversão para cada situação, como pode ser visto no Algoritmo 11, listado como segue:

Algorithm 11 Macro para Formatação de Valores Monetários

```
function FORMAT_MONEY(coluna)
    Remova o prefixo "€"da coluna usando strip_affix
    if coluna contém "M" then
        Remova o sufixo "M"usando strip_affix
        Converta para número e multiplique por 1.000.000
        Retorne o valor como inteiro
    else if coluna contém "K" then
        Remova o sufixo "K"usando strip_affix
        Converta para número e multiplique por 1.000
        Retorne o valor como inteiro
    else
        Converta para número e retorne como inteiro
    end if
end function
```

Para o tratamento da coluna Contract, explicitadas no Quadro 6 e iniciado pela `def start_time_contract`, três macros foram criados:

1. A macro start_time_contract, representada pelo Algoritmo 12, determina o ano de início do contrato:

Algorithm 12 Macro para Determinação do Início do Contrato

```
1: function START_TIME_CONTRACT(coluna)
2:     if coluna contém "~" then
3:         Retorne os primeiros 4 caracteres da coluna como texto (ano)
4:     else if coluna contém "Loan" then
5:         Remova "On Loan"da coluna usando strip_affix
6:         Extraia o ano, convertendo para o formato "Mon DD, YYYY"
7:         Retorne o ano como texto
8:     else if coluna contém "Free" then
9:         Retorne "No Club"
10:    else
11:        Retorne NULL
12:    end if
13: end function
```

2. A macro define_contract_type, representada pelo Algoritmo 13, define o tipo de contrato que o atleta tem com o clube:

Algorithm 13 Macro para Definição do Tipo de Contrato

```
1: function DEFINE_CONTRACT_TYPE(coluna)
2:     if coluna contém "Free" then
3:         Retorne "Free"
4:     else if coluna contém "Loan" then
5:         Retorne "Loan"
6:     else if coluna contém "~" then
7:         Retorne "Contract"
8:     else
9:         Retorne NULL
10:    end if
11: end function
```

3. Por fim, a macro end_contract, representada pelo Algoritmo 14, determina o fim do contrato do atleta com o clube:

Algorithm 14 Macro para Determinação do Fim do Contrato

```
1: function END_CONTRACT(tipo, contrato, empréstimo)
2:     if tipo é "Contract" then
3:         Retorne os últimos 4 caracteres de contrato como ano
4:     else if tipo é "Loan" then
5:         Converta a data de empréstimo para DATE e extraia o ano
6:     else
7:         Retorne "No Club"
8:     end if
9: end function
```

Dessa forma, usando as macros indicadas acima, as colunas `type_of_contract`, `start_year` e `end_year` foram criadas, como é possível ver no Apêndice B.9.

Por fim, o tratamento da coluna Hits, que tem como objetivo converter valores que podem estar expressos em milhares (K) para valores inteiros, foi feita pela `macro convert_hits` e exemplificada pelo Algoritmo 15:

Algorithm 15 Macro para Conversão de Hits

```
1: function CONVERT_HITS(coluna)
2:     if coluna é nula then
3:         Retorne NULL
4:     else if coluna contém "K" then
5:         Remova "K"de coluna usando strip_affix
6:         Multiplique o valor por 1000 e converta para float
7:     else
8:         Converta o valor de coluna diretamente para float
9:     end if
10: end function
```

Com essas macros criados, é necessário aplicá-las na tabela já existente no banco PostgreSQL a partir de um `SELECT * FROM analytics.ELT_clean_fifa_model`, demonstrado no `Código SQL para Aplicação dos Macros de Tratamento`.

Para complementar, testes automatizados foram criados com o objetivo de verificar, em cada nova iteração do projeto, se os dados estão como esperado. O DBT fornece essa funcionalidade por meio do pacote `DBT expectations`, permitindo testar tipos de coluna, valores esperados, modelagens definidas, entre outros aspectos. Esses testes automatizados atuam tanto como um mecanismo de bloqueio em caso de falhas quanto como uma aplicação de boas práticas de software em banco de dados. Como um *fluxo* está sempre recebendo novos dados, a aplicação de testes assegura que nenhuma informação inesperada seja entregue em produção, facilitando o trabalho do engenheiro de dados.

Os testes são detalhados no `arquivo schema.yml`, e são executados automaticamente a cada **DBT run** ou podem ser acionados individualmente pelo comando **DBT test**:

- **Testes de Unicidade e Presença de Valores**:
  - **unique**: Garante que o campo `id` seja único, evitando duplicações na tabela.
  - **not_null**: Assegura que os campos `id`, `name`, `height_cm` e `weight_kg` não contenham valores nulos, garantindo a integridade dos dados pessoais do jogador.

- **Testes de Tipo de Dados**:
  - **DBT_expectations.expect_column_values_to_be_of_type**: Verifica se os valores das colunas correspondem aos tipos de dados esperados:
    - `height_cm`, `weight_kg`, `wf`, `sm`, `ir`: Devem ser do tipo `integer`.
    - `value`, `wage`, `release_clause`: Devem ser do tipo `double precision`.
- **Testes de Valores Aceitos**:
  - **accepted_values**: Garante que a coluna `type_of_contract` contenha apenas os valores pré-definidos: `"Contract"`, `"Loan"` ou `"Free"`. Isso evita inconsistências ou erros na categorização do tipo de contrato dos jogadores.

DBT também permite a geração automatizada de documentação por meio do comando **DBT docs generate**, enquanto o comando **DBT docs serve** abre uma página web interativa com a documentação do projeto, exemplificada pela Figura 7:

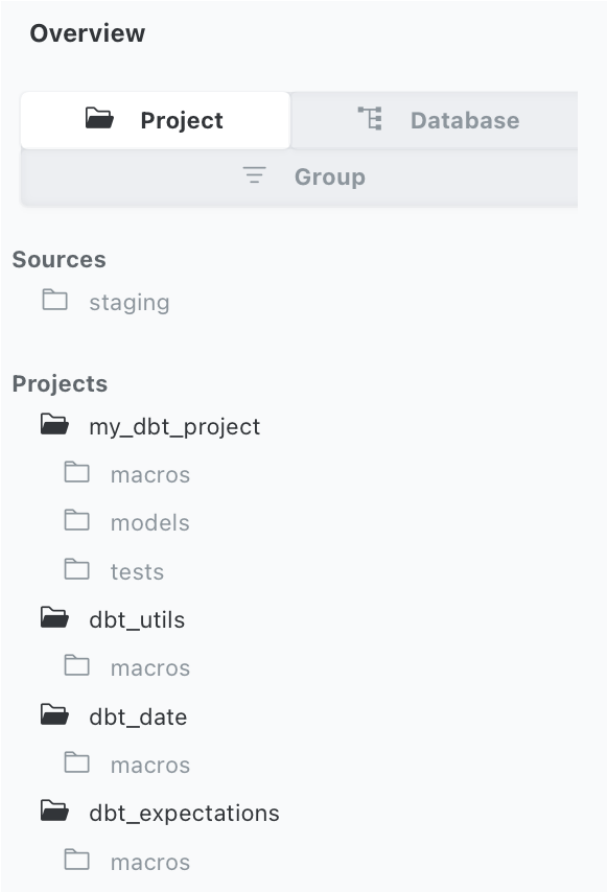

Figura 7: Documentação gerada pelo DBT

4.4 Avaliação das Arquiteturas

A avaliação das arquiteturas de extração de dados ETL e ELT foi feita com base em critérios de escalabilidade, manutenibilidade, desafios, vantagens e cenários ideais de uso,

como indicado na Seção 4.1. A seguir, são apresentados os pontos relevantes para cada abordagem utilizada no projeto:

4.4.1 Escalabilidade

- **ETL (Python - pandas e numpy)**: A escalabilidade da arquitetura ETL desenvolvido em Python é limitada pela capacidade de memória e pela eficiência das operações realizadas. O uso de bibliotecas como `pandas` e `numpy` oferece um bom desempenho para conjuntos de dados pequenos a médios, porém, à medida que o volume de dados cresce, os processos podem enfrentar gargalos, resultando em alto consumo de memória RAM e tempo de processamento elevado. Para lidar com grandes volumes de dados, seria necessário implementar estratégias como:
  - Particionamento dos dados para processamento em lotes menores.
  - Uso de bibliotecas otimizadas para grandes volumes de dados.
  - Migração para frameworks distribuídos e focados em processamento paralelo em clusters, como Apache Spark.
- **ELT (DBT)**: O modelo ELT desenvolvido com DBT apresenta uma escalabilidade significativamente maior, pois delega o processamento ao banco de dados escolhido. No Quadro 12, pode-se ver o tempo de execução de cada um dos fluxos de extração de dados estudados nesse trabalho. Apesar do fluxo ETL ter sido mais rápido, o fluxo ELT contempla testes automatizados e criação de documentação, o que é essencial para governança de dados.

Quadro 12 - Tempos de Execução dos Fluxos ETL e ELT

Fluxo	**Tempo Total**	**Operações Executadas**
ETL	8.10 segundos	Extração, Transformação, Inserção (8.10 s)
ELT	8.37 segundos	Extração e Inserção (6.48 s), Transformação (0.47 s), Testes (1.09 s) e Documentação (0.33 s)

Como as transformações ocorrem diretamente no ambiente SQL, a escalabilidade está atrelada à capacidade do sistema de banco de dados, que pode ser otimizado para grandes volumes. Alguns pontos que podem tornar o DBT mais escalável são:

- Utilização de processamento paralelo e otimizações internas do sistema de banco de dados.
- Suporte a arquiteturas distribuídas, permitindo execução eficiente em bancos na nuvem como BigQuery, Snowflake e Amazon Redshift.

  - Integração com buckets S3 [31], AWS Glue [32] e Athena [33] via `DBT-athena`, possibilitando a execução de transformações diretamente sobre tabelas no Glue Data Catalog.
  - Possibilidade de ajuste de configurações do sistema de banco de dados, como alocação de memória e otimização de índices, para melhorar o desempenho de consultas complexas.

4.4.2 Manutenibilidade

- **ETL (Python - pandas e numpy)**: A manutenibilidade da arquitetura ETL desenvolvida em Python depende diretamente da organização do código e da documentação. Como o código foi estruturado utilizando funções modulares em `pandas` e `numpy`, ele facilita a manutenção e a extensão do processo de ETL, desde que as transformações sejam bem documentadas e que testes sejam aplicados regularmente. No entanto, à medida que mais transformações e fontes de dados são adicionadas, a complexidade do código pode aumentar, impactando a legibilidade e a manutenção a longo prazo. Alguns desafios específicos incluem:
  - Dificuldade de rastrear mudanças e depurar erros devido à natureza imperativa do código.
  - A ausência de um framework declarativo pode tornar mais complexa a reprodução de transformações ou a auditoria dos dados.
  - Dependência de boas práticas de engenharia de software para garantir modularidade e reutilização eficiente do código.
- **ELT (DBT)**: O modelo ELT desenvolvido com DBT possui uma boa manutenibilidade devido à sua estrutura modular e à utilização de macros, que permitem a reutilização de código e a padronização das transformações. DBT se baseia em SQL declarativo, tornando as transformações mais intuitivas e fáceis de compreender. Algumas características que tornam sua manutenção mais eficiente são:
  - Organização em modelos de transformação de dados, permitindo que transformações sejam feitas de forma incremental e reutilizável.
  - Integração nativa com ferramentas de versionamento de código, como Git, facilitando o rastreamento de mudanças.
  - Testes automatizados e validações explícitas no `schema.yml`, que ajudam a identificar inconsistências e prevenir erros antes da execução em produção.
  - Documentação gerada automaticamente pelo DBT, facilitando a rastreabilidade das transformações e o entendimento do fluxo de dados.

4.4.3 Desafios e Vantagens

- **ETL (Python - pandas e numpy)**
  - **Desafios:** No estudo realizado, o conjunto de dados escolhido foi pequeno, pensando em evitar dificuldades que vão além da execução simples do código, como a necessidade de técnicas de paralelização, distribuição de carga e otimização de infraestrutura, além do tempo limitado para conclusão do estudo proposto. Dessa forma, não houve desafios significativos relacionados à limitação de memória ou ao desempenho de processamento. No entanto, ao lidar com conjuntos de dados maiores, como aqueles encontrados em cenários de produção, surgem desafios complexos que exigem essas abordagens avançadas. Para escalar a solução para volumes maiores, seria necessário considerar a paralelização do processamento, a distribuição da carga entre várias máquinas e a otimização da infraestrutura para garantir desempenho satisfatório.
  - **Vantagens:** A arquitetura de ETL em Python, usando bibliotecas como `pandas` e `numpy`, oferece grande flexibilidade e adaptabilidade, permitindo o uso de diversas técnicas de transformação de dados conforme a necessidade. Além disso, a organização do código em células dentro do `Jupyter Notebook` facilita a leitura, personalização e ajustes rápidos durante o desenvolvimento, o que é útil para análises exploratórias com conjuntos de dados pequenos. Essa flexibilidade, contudo, deve ser ampliada e otimizada ao lidar com grandes volumes de dados, o que exigiria um maior investimento em infraestrutura e estratégias de processamento distribuído.
- **ELT (DBT)**
  - **Desafios:** O principal desafio encontrado foi a configuração inicial do DBT e o gerenciamento de dependências. Em um cenário de grandes volumes de dados, seria necessário um banco de dados estruturado e otimizado para permitir operações de transformação de dados de forma eficiente. No caso do estudo realizado, o conjunto de dados pequeno fez com que esses desafios não fossem significativos. Porém, ao expandir para conjuntos de dados maiores, a escolha do sistema de banco de dados e a infraestrutura de execução tornam-se questões relevantes para garantir o desempenho e a escalabilidade do processo.
  - **Vantagens:** DBT delega o processamento ao sistema de banco de dados, aproveitando sua capacidade de execução paralela e otimizações nativas, o que o torna altamente eficiente para grandes volumes de dados. Além disso, ele aprimora a governança de dados, oferecendo testes automatizados, documentação integrada e versionamento de transformações, reduzindo riscos e facilitando a manutenção a longo prazo.

4.4.4 Cenários Ideais

- **ETL (Python - pandas e numpy)**: Considerando o conjunto de dados de tamanho reduzido escolhido e as transformações simples que o conjunto de dados requisitava, o fluxo ETL serve muito bem para o conjunto de dados em questão. Ele é ideal para cenários em que os dados não são volumosos, a transformação requer mais flexibilidade e customização, e onde a simplicidade do código é mais valorizada do que a escalabilidade. Para projetos de menor escala ou quando se tem controle total sobre o ambiente de execução, o uso de Python é vantajoso devido à facilidade de implementação e à vasta quantidade de bibliotecas disponíveis para processamento e análise de dados.

- **ELT (DBT)**: O uso de DBT pode ser considerado menos relevantes para projetos menores, mas em cenários de maior escala e complexidade de dados, ele se torna uma escolha robusta. O DBT é ideal quando há necessidade de transformações que podem ser executadas diretamente no banco de dados, aproveitando sua capacidade de processamento paralelo e escalabilidade. Além disso, em ambientes com grandes volumes de dados, como aqueles encontrados em plataformas de nuvem (BigQuery, Redshift, Snowflake), o DBT pode lidar de maneira eficiente com grandes volumes de dados sem sobrecarregar as máquinas locais. Em resumo, o DBT faz com que o processo de governança dos dados seja mais suave, com mais controle sobre o versionamento das transformações, validações automáticas e documentação gerada automaticamente, garantindo uma maior rastreabilidade e auditoria ao longo do fluxo de dados [34].

4.4.5 Estudo sobre Tendências Futuras

Com o avanço contínuo no volume e na complexidade dos dados, as abordagens de ETL e ELT passam constantemente por transformações significativas, com o objetivo de atender às novas demandas do mercado, que procuram sempre aprimorar pontos como escalabilidade, desempenho e governança. Algumas das principais tendências que estão moldando o futuro dessas tecnologias são:

- **Automação e IA nos Pipelines**: A automação das transformações de dados está se tornando cada vez mais inteligente, com a integração de IA e aprendizado de máquina, permitindo que os fluxos de dados se tornem mais dinâmicos, aprendam com experiências passadas e se ajustem automaticamente para otimizar a qualidade e a eficiência das transformações [35].

- **Pipelines de Dados em Tempo Real**: A demanda por dados em tempo real já é uma realidade, com ferramentas de visualização como Grafana [36] sendo cada vez

mais requisitadas em contextos de governança operacional, e ETL/ELT estão evoluindo para atender essa necessidade com processamentos de baixa latência. Tecnologias como Apache Kafka estão ganhando destaque, permitindo a inserção e transformação contínua de dados à medida que são gerados. Espera-se que a integração de fluxos e essas plataformas seja ainda mais harmoniosa, proporcionando uma experiência fluida e eficiente [15].

- **Computação em Nuvem e Arquitetura Serverless**: A adoção de soluções baseadas em nuvem e arquiteturas serverless, como o AWS Lambda [37], tem se tornado parte essencial do planejamento estratégico das empresas em termos de escalabilidade. Essas tecnologias possibilitam um processamento mais ágil, escalável e econômico, ao mesmo tempo em que reduzem a necessidade de manutenção de infraestrutura. Com o tempo, espera-se uma integração cada vez maior entre os fluxos de dados e os provedores de nuvem, promovendo maior flexibilidade e eficiência nos processos.

- **ELT e Governança de Dados**: O processo de ELT está se tornando mais abrangente, com ferramentas como DBT permitindo não apenas a transformação de dados, mas também a governança completa do fluxo. A rastreabilidade, o versionamento de transformações e a documentação automatizada já são fundamentais para garantir conformidade regulatória e transparência, e cada vez mais esses aspectos serão considerados pré-requisitos.

- **Ferramentas Low-code/No-code**: As plataformas de low-code e no-code estão em alta e ganhando cada vez mais popularidade, democratizando a criação de fluxos de dados e tornando-os acessíveis para profissionais de outros focos e áreas, possibilitando que eles construam e mantenham fluxos de dados de forma eficiente. Essa tendência está tornando o processo de transformação de dados mais acessível e ágil, reduzindo a dependência de equipes altamente especializadas.

As tendências apontadas circulam em torno de três pontos principais: automação, escalabilidade e integração com tecnologias emergentes. Isso não apenas facilitará a gestão e a análise de dados em tempo real e em grande escala, mas também tornará o processo mais acessível e eficiente para empresas de todos os tamanhos. O caminho à frente é claro: transformar dados em conhecimento de forma rápida, segura e inteligente.

4.4.6 Conclusão do Estudo de Caso

Neste capítulo, foram exploradas as arquiteturas ETL e ELT em um contexto prático, com foco na comparação entre elas em termos de desempenho, escalabilidade, manutenção e adaptação a diferentes cenários. Ao longo do capítulo, foi descrito detalhadamente o

conjunto de dados utilizado, abrangendo cada uma das transformações necessárias durante a etapa de limpeza. A análise considerou tanto as necessidades de preparação dos dados como as diferenças nas abordagens de transformação das etapas dos pipelines ETL e ELT.

A arquitetura ETL, implementada com Python e utilizando as bibliotecas pandas e numpy, foi detalhadamente abordada, evidenciando o fluxo de extração, transformação e carregamento dos dados. As transformações realizadas foram discutidas passo a passo, incluindo os procedimentos de limpeza e validação dos dados, destacando as limitações existentes dessa arquitetura. A arquitetura ELT, baseada no DBT, foi igualmente detalhada, com ênfase no fluxo de extração e carregamento dos dados, seguido pelas transformações realizadas diretamente no sistema de banco de dados.

As diferenças entre as etapas de transformação nos pipelines ETL e ELT foram exemplificadas de forma clara. No pipeline ETL, as transformações ocorrem antes do carregamento, o que exige que o processo de transformação seja mais intensivo e demande mais recursos de computação local. Já a arquitetura ELT, as transformações são realizadas após o carregamento, permitindo que o sistema banco de dados aproveite seus próprios recursos para realizar as transformações de forma mais eficiente.

Além disso, os aspectos de avaliação de cada arquitetura foram amplamente comentados, levando em consideração métricas como escalabilidade, manutenibilidade, confiabilidade dos dados e flexibilidade. A arquitetura ETL considerada nesse estudo foi analisada principalmente sob a perspectiva de sua eficiência em cenários menores, com o uso de Python proporcionando uma boa solução para volumes reduzidos de dados, mas com limitações em termos de escalabilidade. O DBT, por sua vez, se destacou pela facilidade de manutenção, especialmente quando se trata de governança de dados, automação de testes e geração de documentação.

Em síntese, o estudo de caso detalhou as principais diferenças entre as arquiteturas ETL e ELT, destacando as vantagens de cada abordagem em diferentes contextos de uso. A comparação entre as arquiteturas foi feita com base em uma análise das transformações realizadas nos dados, dos recursos necessários e dos impactos de cada arquitetura no desempenho geral do processo de integração de dados. No entanto, a escolha de um conjunto de dados de volume reduzido impôs certas limitações ao estudo, restringindo a avaliação de aspectos como desempenho em larga escala e impacto da escalabilidade em cenários mais complexos. Futuras pesquisas poderiam explorar essas variáveis com conjuntos de dados mais robustos, permitindo uma análise mais abrangente das diferenças entre as arquiteturas ETL e ELT. Além disso, apesar dos resultados apresentados nesse capítulo sejam relevantes e resultantes de um estudo pioneiro do ponto de vista prático e experimental, é essencial reconhecer que tais resultados são apenas conclusivos para o conjunto de dados do estudo de caso proposto.

5 Conclusão

5.1 Considerações Finais

O estudo de caso abordou duas arquiteturas comuns de fluxo de dados no contexto de pipelines de dados, implementando os modelos ETL e ELT para um conjunto de dados extraído do Kaggle, proveniente do jogo FIFA 21. O fluxo ETL foi implementado em Python, utilizando bibliotecas como pandas e numpy para a transformação de dados e inserção diretamente em um banco PostgreSQL. O fluxo ELT, por sua vez, utilizou DBT para realizar a transformação dos dados diretamente no banco de dados PostgreSQL, após a extração e inserção dos dados.

A arquitetura ETL demonstrou ser mais adequada para conjuntos de dados de pequeno porte e em contextos onde a etapa de transformação não exige operações computacionalmente intensivas ou muito complexas, como agregações de dados ou transformações que envolvem grandes volumes de informações simultâneas. Tais operações são aquelas que exigem cálculos pesados, manipulação de grandes volumes de dados ou transformações que dependem de múltiplas fontes de dados, como junções (joins) complexas, agregações múltiplas ou cálculos em tempo real. Essas operações não foram consideradas no estudo de caso, o que limitou a avaliação do ETL em termos de complexidade e escalabilidade. A implementação em Python, ao usar bibliotecas como pandas e numpy, possibilitou um maior controle sobre cada etapa do processo, permitindo uma filtragem e transformação de dados de forma detalhada. Isso trouxe flexibilidade, pois o código pode ser facilmente ajustado para diferentes tipos de análise, e simplicidade, já que as operações puderam ser feitas de forma sequencial e transparente. Contudo, como o conjunto de dados utilizado foi pequeno e as transformações não exigiram grandes complexidades, a arquitetura ETL não foi desafiada em termos de processamento e escalabilidade, o que impediu uma avaliação mais aprofundada desses aspectos.

Por outro lado, o pipeline ELT, utilizando DBT, é especialmente vantajoso em cenários com grandes volumes de dados e transformações complexas, aproveitando a infraestrutura do banco de dados para realizar as transformações de forma eficiente em grande escala. No entanto, devido ao uso de um conjunto de dados pequeno no estudo de caso, não foi possível explorar todo o potencial do DBT em termos de escalabilidade e processamento paralelo. Mesmo assim, foi possível evidenciar as vantagens do DBT em aspectos de governança de dados, como testes automatizados, geração de documentação e o uso de macros gerando códigos modularizados, que são práticas comuns em Engenharia de Software.

As limitações deste estudo estão principalmente relacionadas à escolha de um conjunto de dados pequeno e à ausência de tecnologias de Big Data, que seriam mais apropriadas para lidar com volumes maiores de dados. Além disso, o uso de um fluxo simples e

com baixo volume de dados não permitiu observar o desempenho das abordagens quando aplicadas a conjuntos de dados em larga escala.

O conjunto de dados escolhido para esse estudo teve como objetivo realizar várias transformações antes e depois da geração da base de dados final. Para esse fim, a base de dados usada foi satisfatória. No entanto, a necessidade de um volume maior de dados só foi identificada com a coleta dos resultados obtidos, não havendo tempo hábil para realizar testes adicionais. Além disso, a escolha do tamanho da base de dados foi influenciada pelo tempo disponível para realização do estudo proposto.

As tendências futuras em integração e transformação de dados apontam para a adoção de arquiteturas híbridas de ETL/ELT, integração com IA e Machine Learning, e automação com ferramentas como DBT e Apache Airflow [38]. A ascensão do DataOps [39], que promove a colaboração e automação de fluxos de dados, também traz avanços na eficiência dos pipelines. Além disso, tecnologias emergentes como Data Lakehouses [40], processamento em tempo real com Apache Kafka e plataformas serverless estão transformando a gestão e o processamento de grandes volumes de dados, oferecendo soluções mais flexíveis e escaláveis.

Em resumo, este estudo de caso oferece uma perspectiva sobre as arquiteturas ETL e ELT, mas também aponta para futuras melhorias no uso de arquiteturas distribuídas, tecnologias avançadas e maior integração entre diferentes fontes de dados, com o objetivo de otimizar a escalabilidade e a performance dos processos de transformação e integração de dados.

5.2 Limitações

Embora as arquiteturas ETL e ELT apresentados tenham atendido ao escopo inicial do estudo de caso, que teve como objetivo primário realizar várias transformações antes e depois da geração da base de dados final, algumas limitações se tornam evidentes devido ao contexto e à escolha do conjunto de dados usado:

- **Tamanho do conjunto de dados**: O conjunto de dados utilizado é de tamanho reduzido, o que limita a capacidade de melhor investigar as vantagens de escalabilidade e desempenho das ferramentas utilizadas, especialmente no caso do DBT. Em cenários de Big Data, onde o volume de dados é significativamente maior, as vantagens de ferramentas como DBT (com seu processamento paralelo, otimização no banco de dados e integração de dados de origens distintas) se tornam mais evidentes, o que não foi explorado neste estudo.

- **Ausência de Ferramentas de Big Data**: A escolha de um conjunto de dados pequeno e o uso de ferramentas convencionais limitaram a demonstração da escalabilidade do fluxo. Em ambientes reais, onde grandes volumes de dados exigem

processamento distribuído, frameworks como Hadoop e Spark são fundamentais para garantir eficiência.

- **Limitações do Conjunto de Dados acerca de Integração de Dados**: Embora DBT seja uma ferramenta robusta para transformações em larga escala e integração de dados diversos, o conjunto de dados utilizado no estudo diz respeito a uma única aplicação e é proveniente de uma única fonte de dados, não permitindo explorar todo o seu potencial de integração de dados. Recursos como modelagem avançada, integração de múltiplas fontes e otimização de fluxos complexos ficaram subaproveitados devido à simplicidade do conjunto de dados e das transformações aplicadas. O fluxo ELT poderia ser ajustado para:
  - Aproveitar seu suporte a arquiteturas distribuídas, possibilitando execução eficiente em bancos de dados na nuvem como BigQuery, Snowflake e Amazon Redshift.
  - Integrar-se a buckets S3, AWS Glue e Athena via `DBT-athena`, permitindo a execução de transformações diretamente sobre tabelas no Glue Data Catalog.
- **Simplicidade do fluxo**: A escolha de um fluxo simples e com baixo volume de dados não permite demonstrar plenamente o potencial das ferramentas. Em cenários mais complexos, onde o tratamento de dados exige maior governança, orquestração de múltiplos processos e integração com várias fontes, a utilização de ferramentas como DBT seria mais justificada, e a integração com plataformas de Big Data seria uma escolha mais eficaz e escalável.

As limitações identificadas refletem as escolhas feitas no desenvolvimento dos fluxos e indicam oportunidades de melhoria em escalabilidade, flexibilidade e desempenho. Estudos futuros podem superar essas restrições adotando tecnologias mais avançadas ou arquiteturas distribuídas que permitam grandes volumes de dados e maior eficiência no processamento, maximizando o potencial das ferramentas utilizadas.

5.3 Trabalhos Futuros

Com base nas limitações discutidas na Seção 5.2 e no aprendizado obtido na Seção 4.4, durante a avaliação das arquiteturas, bem como nas tendências futuras analisadas na Seção 4.4.5, diversas melhorias e expansões podem ser realizadas para aumentar a eficiência, escalabilidade e robustez dos fluxos de ETL e ELT. Algumas das direções possíveis para trabalhos futuros são:

- **Escalabilidade e Big Data**: Para explorar as vantagens de ferramentas como DBT em cenários de Big Data, seria interessante utilizar conjuntos de dados maiores, que

permitam investigar o poder de processamento paralelo e a otimização de consultas em bancos de dados de grande escala. A integração com ferramentas como Hadoop ou Apache Spark também pode ser uma abordagem válida para lidar com grandes volumes de dados e processos distribuídos, permitindo que o fluxo se torne mais eficiente e escalável.

- **Foco em Integração de Dados**: Um próximo passo seria expandir o fluxo para integrar dados vindos de diversas fontes, tanto relacionais quanto não relacionais (NoSQL, APIs, Excel, etc). O DBT, apesar de ser uma excelente ferramenta para modelagem de dados e integração com bancos de dados SQL, poderia ser melhor aproveitado em cenários em que múltiplas fontes de dados precisem ser integradas.

- **Automação e Orquestração Completa**: A automação e orquestração dos fluxos, principalmente em estudos maiores e mais complexos, seriam essenciais para garantir o gerenciamento eficiente de processos. A construção de um fluxo com Airflow poderia ser utilizado para agendar, monitorar e garantir a execução eficiente dos fluxos, além de permitir a reexecução de falhas e a rastreabilidade de todas as operações realizadas.

- **Análise de Dados em Tempo Real**: Para melhorar a flexibilidade do fluxo, seria interessante explorar ferramentas que permitem o processamento de dados em tempo real, como Apache Kafka. Isso permitiria que os dados fossem transformados e analisados assim que fossem coletados, ao invés de depender de operações em batch, o que seria mais adequado para cenários que exigem uma resposta imediata ou integração contínua com sistemas de produção.

- **Cloud Platform**: A migração para uma plataforma de nuvem como AWS ou Azure permitiria escalar os fluxos de maneira mais eficiente, utilizando serviços gerenciados para processamento de dados como AWS Glue. Além disso, essas plataformas fornecem recursos adicionais de integração e governança de dados que podem aprimorar o desempenho e a flexibilidade do fluxo.

- **Aprimoramento da Governança de Dados**: A governança de dados pode ser aprimorada com o uso de técnicas mais avançadas de versionamento e auditoria das transformações aplicadas. O DBT já oferece funcionalidades importantes nesse aspecto, mas, além de testes automatizados e documentação, o aspecto de governança que o DBT fornece não foi explorado totalmente devido à simplicidade do fluxo criado. A integração com ferramentas e features adicionais pode garantir rastreabilidade, documentação e validação de dados seria uma área interessante de desenvolvimento.

Essas melhorias pontuadas podem ajudar a criar estudos futuros com maior abrangência e impacto, além de potencializar o uso das ferramentas escolhidas, permitindo a utilização dessas soluções em cenários de dados mais complexos e de maior volume.

Referências

[1] Natalia Miloslavskaya e Alexander Tolstoy. "Big Data, Fast Data and Data Lake Concepts". Em: *Procedia Computer Science* 150 (2019).

[2] W.H. Inmon. *Data Architecture: A Primer for the Data Scientist*. Academic Press, 2018.

[3] Edward Manopo Haryono et al. "Comparison of the E-LT vs ETL Method in Data Warehouse Implementation: A Qualitative Study". Em: *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*. IEEE. 2020.

[4] Bharat Singhal e Alok Aggarwal. "ETL, ELT and Reverse ETL: A Business Case Study". Em: *2022 Second International Conference on Advanced Technologies in Intelligent Control, Environment, Computing and Communication Engineering (ICATIECE)*. IEEE. 2022.

[5] Ralph Kimball et al. *The Data Warehouse Lifecycle Toolkit*. John Wiley & Sons, 1996.

[6] Michael Chen. *What is Big Data?* Disponível em: https://www.oracle.com/big-data/what-is-big-data/. Acesso em: 07 mar. 2025. Set. de 2024.

[7] Fishtown Analytics. *dbt: Analytics Engineering for Modern Data Stack*. Rel. técn. dbt Labs, 2021.

[8] Ralph Kimball e Margy Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. John Wiley & Sons, 2013.

[9] Wikipedia contributors. *Cloud database*. Disponível em: https://en.wikipedia.org/wiki/Cloud_database. Acesso em: 14 mar. 2025. 2024.

[10] AnHai Doan, Alon Halevy e Zachary Ives. *Principles of Data Integration*. San Francisco: Morgan Kaufmann, 2012.

[11] Matthew Kosinski. *What is a Data Lake?* Disponível em: https://www.ibm.com/think/topics/data-lake. Acesso em: 05 mar. 2025. 2025.

[12] Denis Henrique Pazini da Silva et al. "DATA LAKE: Suas Funcionalidades e Aplicações". Em: *Informação e Sociedade: Estudos* 21.1 (jun. de 2024). DOI: 10.31510/infa.v21i1.1960. URL: https://doi.org/10.31510/infa.v21i1.1960.

[13] Matei Zaharia et al. "Apache Spark: A Unified Engine for Big Data Processing". Em: *Communications of the ACM*. Vol. 59. 11. ACM. 2016, pp. 56–65.

[14] Apache Software Foundation. *Apache Kafka Documentation*. 2025. URL: https://kafka.apache.org/documentation/.

[15] Vu Trinh. *How does Uber build real-time infrastructure to handle petabytes of data every day?* 2024. URL: https://blog.det.life/how-does-uber-build-real-time-infrastructure-to-handle-petabytes-of-data-every-day-ddf5fe9b5d2c.

[16] Darren Hickling. *How to Use AWS Glue Catalog to Empower Your Modern Data Governance*. Jul. de 2023. URL: https://www.contino.io/insights/aws-glue-catalog.

[17] IBM. *IBM InfoSphere DataStage: Data Integration for a Trusted Data Foundation*. Rel. técn. IBM Corporation, 2010.

[18] SAS Institute. *SAS Data Integration Studio: An Overview*. Rel. técn. SAS Institute, 2012.

[19] Microsoft. *SQL Server Integration Services (SSIS) Overview*. Rel. técn. Microsoft Corporation, 2019.

[20] Apache Software Foundation. *Introduction to Apache NiFi: Dataflow Automation*. Rel. técn. Apache Software Foundation, 2019.

[21] Python Software Foundation. *Python Documentation*. 2020. URL: https://docs.python.org.

[22] Google. *BigQuery Documentation*. 2024. URL: https://cloud.google.com/bigquery/docs.

[23] Amazon Web Services. *Amazon Redshift: Data Warehouse in the Cloud*. Rel. técn. AWS, 2020.

[24] Snowflake Inc. *Snowflake: The Data Cloud*. Rel. técn. Snowflake, 2022.

[25] Chris Garzon. *Why Python Is Still the Most Important Language for Data Engineers*. Mar. de 2025. URL: https://dataengineeracademy.com/module/why-python-is-still-the-most-important-language-for-data-engineers/.

[26] Dhamotharan Seenivasan. "ETL vs ELT: Choosing the right approach for your data warehouse". Em: *International Journal for Research Trends and Innovation* (2022), pp. 110–122.

[27] Sebastian Pulkka. "The Modernization Process of a Data Pipeline". Diss. de mestr. Åbo Akademi University, Faculty of Science e Engineering, 2023.

[28] Regdate Software. *Knowledge Base of Relational and NoSQL Database Management Systems*. Accessed: 2025-03-16. 2025. URL: https://db-engines.com/en/ranking_trend.

[29] dbt Labs. “DBT Labs Builds Momentum as the Industry Standard for Data Transformation”. Em: *PR Newswire* (ago. de 2023). Accessed: 2025-03-16. URL: https://www.getdbt.com/blog/dbt-labs-builds-momentum-as-the-industry-standard-for-data-transformation.

[30] dbt Labs. *Using Jinja in dbt*. Acessado em: 20 mar. 2025. 2024. URL: https://docs.getdbt.com/docs/build/jinja-macros.

[31] Amazon Web Services. *Amazon S3 - Scalable Storage in the Cloud*. Accessed: 2025-03-12. 2024. URL: https://aws.amazon.com/s3/.

[32] Amazon Web Services. *AWS Glue - Fully Managed ETL Service*. Accessed: 2025-03-12. 2024. URL: https://aws.amazon.com/glue/.

[33] Amazon Web Services. *Amazon Athena - Serverless Interactive Query Service*. Accessed: 2025-03-12. 2024. URL: https://aws.amazon.com/athena/.

[34] Atlan. *dbt Data Governance: How to Enable Active Data Governance for Your dbt Assets*. Accessed: 2025-03-21. 2024. URL: https://atlan.com/dbt-data-governance/.

[35] Akash Takyar. *AI in data integration: Types, challenges, key AI techniques and future*. 2024. URL: https://www.leewayhertz.com/ai-in-data-integration/.

[36] Grafana Labs. *Grafana - Open Source Analytics Monitoring*. Accessed: 2025-03-12. 2024. URL: https://grafana.com/.

[37] Amazon Web Services. *Serverless Computing with AWS Lambda*. 2023. URL: https://aws.amazon.com/lambda/.

[38] Apache Software Foundation. *Apache Airflow*. Acesso em: 19 mar. 2025. 2025. URL: https://airflow.apache.org/.

[39] Andy Palmer. *DataOps: Agile Data Engineering*. https://www.dataopsmanifesto.org/. 2022.

[40] Forbes. *Data Lakehouse*. 2024. URL: https://www.forbes.com/councils/forbestechcouncil/2024/09/06/the-power-of-the-data-lakehouse-shaping-the-future-of-analytics-and-machine-learning/.

APÊNDICES

A Apêndice 1: Funções de Transformação em Python

A.1 ft_to_cm

```
def ft_to_cm(x):
    if "'" in x:
        parts = x.split("'")
        feet = int(parts[0])
        inches = int(parts[1].replace('"', '')) if parts[1] else
   0
        return int(round((feet * 30.48) + (inches * 2.54), 0))

    return int(x.strip("cm"))

df["height"]=df["height"].apply(ft_to_cm)
```

A.2 lbs_to_kg

```
def lbs_to_kg(x):
    if "lbs" in x:
        lbs = x.replace("lbs", "")
        kilograms = round(int(lbs) / 2.2, 0)
        return int(kilograms)
    else:
        x_strip = x.strip("kg")
        return int(x_strip)

df["weight"]=df["weight"].apply(lbs_to_kg)
```

A.3 money

```
def money(x):
    if "€" in x:
        x = x.replace("€", "")
    if "M" in x:
        x = x.replace("M", "")
        return int(float(x) * 1_000_000)
    elif "K" in x:
        x = x.replace("K", "")
        return int(float(x) * 1_000)
```

```
    return int(x)

df["value"] = df["value"].apply(money) / 1_000_000
df["wage"] = df["wage"].apply(money)
df["release_clause"] =
            df["release_clause"].apply(money) / 1_000_000

df.rename(columns={
    "value": "values_in_euro_million",
    "wage": "wage_in_euros",
    "release_clause": "release_clause_in_euro_million"
}, inplace=True)
```

A.4 remove_star_from_columns

```
def remove_star_from_columns(df, columns):
    for col in columns:
        df[col] = df[col].str.replace("*", "", regex=False)
    return df

df = remove_star_from_columns(df, ['wf','sm','ir'])
```

A.5 contract_type

```
def contract_type(x):
    if "Free" in x:
        return "Free"
    if "Loan" in x:
        return "Loan"
    if "~" in x:
        return "Contract"
    else:
        return pd.NA
```

A.6 start_contract_start

```
def start_time_contract(x):
    if "~" in x:
        return  int(x[:4])
    if "Loan" in x:
```

```
        x=x.strip(" On Loan")
        x= datetime.strptime(x, "%b %d, %Y")
        return x.date()
    else:
        return "No Club"

df["type_of_contract"] = df["contract"].apply(type)
df["start_year"] = df["contract"].apply(start_time_contract)
```

A.7 end_time_contract

```
def end_time_contract(type, contract, loan):
    if type == "Contract":
        return int(contract[-4:])
    if type == "Loan":
        loan = datetime.strptime(loan, "%b %d, %Y")
        return loan.date()
    else:
        return "No Club"

df["end_year"] = df.apply(lambda row: end_time_contract(row["
    type_of_contract"],row["contract"],row["loan_date_end"]), axis
    =1)

# coluna loan_date_end desnecessaria agora
df = df.drop(columns=["loan_date_end"])
```

A.8 convert_hits

```
def convert_hits(x):
    if pd.isna(x):
        return np.nan
    elif 'K' in str(x):
        return float(x[:-1])
    else:
        return float(x)

# coluna hits desnecessaria agora
df.drop(columns=['hits'], inplace=True)

```

```
12 # preenchendo os valores vazios pela media de chutes a gol
13 df.fillna({'hits_in_k': df['hits_in_k'].mean()}, inplace=True)
```

B Apêndice 2: Macros no DBT e Códigos SQL

B.1 macro ft_to_cm

```
{% macro ft_to_cm(column) %}
    CASE
        WHEN {{ column }} LIKE '%''%' THEN
            CAST(
                (
                    (CAST(SPLIT_PART({{ strip_affix(column, 'cm') }}, '''', 1) AS FLOAT) * 30.48) +
                    (CAST(REPLACE(SPLIT_PART({{ strip_affix(column, 'cm') }}, '''', 2), '"', '') AS FLOAT) * 2.54)
                ) AS NUMERIC
            )::INT
        ELSE
            CAST({{ column }} AS INT)
    END
{% endmacro %}
```

B.2 macro lbs_to_kg

```
{% macro lbs_to_kg(column) %}
    CASE
        WHEN {{ column }} LIKE '%lbs%' THEN
            CAST(
                (
                    (CAST({{ strip_affix(column, 'lbs') }} AS FLOAT) / 2.2)
                ) AS NUMERIC
            )::INT
        ELSE
            CAST({{ column }} AS INT)
    END
{% endmacro %}
```

B.3 macro format_money

```
{% macro format_money(column) %}
CASE
```

```
    WHEN {{ column }} LIKE '€%M' THEN
        CAST({{ strip_affix(strip_affix(column, '€'), 'M') }} AS
   FLOAT) * 1000000
    WHEN {{ column }} LIKE '€%K' THEN
        CAST({{ strip_affix(strip_affix(column, '€'), 'K') }} AS
   FLOAT) * 1000
    ELSE
        CAST({{ strip_affix(column, '€') }} AS FLOAT)
END
{% endmacro %}
```

B.4 macro start_time_contract

```
{% macro start_time_contract(column) %}
CASE
    WHEN {{ column }} LIKE '%~%' THEN
        CAST(SUBSTRING({{ column }}, 1, 4) AS TEXT)
    WHEN {{ column }} LIKE '%Loan%' THEN
        CAST(
            EXTRACT(YEAR FROM TO_DATE({{ strip_affix(column, ' On
    Loan') }}, 'Mon DD, YYYY'))
            AS TEXT
        )
    WHEN {{ column }} LIKE '%Free%' THEN
        'No Club'
    ELSE NULL
END
{% endmacro %}
```

B.5 macro define_contract_type

```
{% macro define_contract_type(column) %}
CASE
    WHEN {{ column }} LIKE '%Free%' THEN 'Free'
    WHEN {{ column }} LIKE '%Loan%' THEN 'Loan'
    WHEN {{ column }} LIKE '%~%' THEN 'Contract'
    ELSE NULL
END
{% endmacro %}
```

B.6 macro end_contract

```
{% macro end_contract(type, contract, loan) %}
CASE
    WHEN {{ type }} = 'Contract' THEN
        CAST(SUBSTRING({{ contract }}, LENGTH({{ contract }}) -
   3, 4) AS TEXT)
    WHEN {{ type }} = 'Loan' THEN
        CAST(
            EXTRACT(YEAR FROM TO_DATE({{ loan }}, 'Mon DD, YYYY')
   ) AS TEXT
        )
    ELSE 'No Club'
END
{% endmacro %}
```

B.7 macro remove_newlines

```
{% macro remove_newlines(column_name) %}
    TRIM(REGEXP_REPLACE({{ column_name }}, '\\n', '', 'g'))
{% endmacro %}
```

B.8 macro convert_hits

```
{% macro convert_hits(column) %}
CASE
    WHEN {{ column }} IS NULL THEN NULL
    WHEN {{ column }} LIKE '%K' THEN
        CAST({{ strip_affix(column, 'K') }} AS FLOAT) * 1000
    ELSE CAST({{ column }} AS FLOAT)
END
{% endmacro %}
```

B.9 ELT_clean_fifa_model.sql, código que aplica os macros na tabela já existente no PostgresQL

```
{{ config(materialized='table') }}

WITH cleaned_data AS (
```

```
4     SELECT
5         *
6         , {{ ft_to_cm('"height"') }} AS "height_cm"
7         , {{ lbs_to_kg('"weight"') }} AS "weight_kg"
8         , {{ format_money('"value"') }} / 1000000 AS "
  value_in_euro_millions"
9         , {{ format_money('"wage"') }} AS "wage_in_euros"
10        , {{ format_money('"release_clause"') }} / 1000000 AS "
  release_clause_in_euro_millions"
11        , {{ end_contract('"type_of_contract"', '"contract"', '"
  loan_date_end"') }} AS "end_year"
12    FROM (
13        SELECT
14            "ID" AS "id",
15            "Name" AS "name",
16            "Nationality" AS "nationality",
17            "Age" AS "age",
18            "↓OVA" AS "ova",
19            "Wage" AS "wage",
20            "Value" AS "value",
21            "Release Clause" AS "release_clause",
22            {{ remove_newlines('"Club"') }} AS "club",
23            "POT" AS "pot",
24            "Contract" AS "contract",
25            {{ define_contract_type('"Contract"') }} AS "
  type_of_contract",
26            {{ start_time_contract('"Contract"') }} AS "
  start_year",
27            "Positions" AS "positions",
28            {{ strip_affix('"Height"', 'cm') }} AS "height",
29            {{ strip_affix('"Weight"', 'kg') }} AS "weight",
30            "Preferred Foot" AS "preferred_foot",
31            "BOV" AS "bov",
32            "Best Position" AS "best_position",
33            "Joined" AS "joined",
34            "Loan Date End" AS "loan_date_end",
35            CAST("Attacking" / 5 AS INT) AS "attacking",
36            CAST("Skill" / 5 AS INT) AS "skill",
37            CAST("Movement" / 5 AS INT) AS "movement",
38            CAST("Power" / 5 AS INT) AS "power",
39            CAST("Mentality" / 6 AS INT) AS "mentality",
```

```
            CAST("Defending" / 3 AS INT) AS "defending",
            CAST("Goalkeeping" / 5 AS INT) AS "goalkeeping",
            CAST({{ strip_affix('"W/F"', '*') }} AS INT) AS "wf",
            CAST({{ strip_affix('"SM"', '*') }} AS INT) AS "sm",
            "A/W" AS "aw",
            "D/W" AS "dw",
            CAST({{ strip_affix('"IR"', '*') }} AS INT) AS "ir",
            {{ convert_hits('"Hits"') }} AS "hits"
        FROM {{ source('staging', 'staging_fifa_model_elt') }}
    ) raw_data
)

SELECT * FROM cleaned_data
```

B.10 Arquivo schema.yml, onde estão centralizados os testes automatizados

```
version: 2

models:
  - name: ELT_clean_fifa_model
    description: "A starter DBT model with additional data quality tests"
    columns:
      - name: id
        description: "The primary key for this table"
        tests:
          - unique
          - not_null

      - name: name
        description: "Player name"
        tests:
          - not_null

      - name: height_cm
        description: "Height in centimeters"
        tests:
          - not_null
          - DBT_expectations.expect_column_values_to_be_of_type:
              column_type: integer
```

```
- name: weight_kg
  description: "Weight in kilograms"
  tests:
    - not_null
    - DBT_expectations.expect_column_values_to_be_of_type:
        column_type: integer

- name: wf
  description: "Weak Foot rating"
  tests:
    - DBT_expectations.expect_column_values_to_be_of_type:
        column_type: integer

- name: sm
  description: "Skill Moves rating"
  tests:
    - DBT_expectations.expect_column_values_to_be_of_type:
        column_type: integer

- name: ir
  description: "International Reputation rating"
  tests:
    - DBT_expectations.expect_column_values_to_be_of_type:
        column_type: integer

- name: value_in_euro_millions
  description: "Player market value in millions of euros"
  tests:
    - DBT_expectations.expect_column_values_to_be_of_type:
        column_type: double precision

- name: wage_in_euros
  description: "Player weekly wage in euros"
  tests:
    - DBT_expectations.expect_column_values_to_be_of_type:
        column_type: double precision

- name: release_clause_in_euro_millions
  description: "Player release clause in millions of euros"
  tests:
```

```
    - DBT_expectations.expect_column_values_to_be_of_type:
        column_type: double precision

- name: type_of_contract
  description: "Type of contract (Contract, Loan, Free)"
  tests:
    - accepted_values:
        values: ["Contract", "Loan", "Free"]
```