INSTITUTO FEDERAL
Rondônia

MINISTÉRIO DA EDUCAÇÃO
Secretaria de Educação Profissional e Tecnológica
Instituto Federal de Educação, Ciência e Tecnologia de Rondônia

Campus Vilhena

Coordenação do Curso Superior em Tecnologia em Análise e Desenvolvimento de Sistemas

LUCAS FERREIRA RODRIGUES

Desenvolvimento da biblioteca Python AutoDBLoader

VILHENA - RO

2025

LUCAS FERREIRA RODRIGUES

Desenvolvimento da biblioteca Python AutoDBLoader

Artigo entregue como Trabalho de Conclusão de Curso apresentado ao Instituto Federal de Educação, Ciência e Tecnologia de Rondônia (IFRO), Campus Vilhena, como requisito parcial para obtenção do grau de Tecnólogo, junto ao Curso Superior em Tecnologia em Análise e Desenvolvimento de Sistemas, sob a orientação do professor José Lucas Brandão Montes.

VILHENA - RO

2025

Ficha catalográfica elaborada pelo Sistema Gerador de Ficha Catalográfica do IFRO.

Rodrigues, Lucas Ferreira.
Desenvolvimento da biblioteca Python AutoDBLoader / Lucas Ferreira Rodrigues. - Vilhena, 2025.
29 f.

Orientador(a): Prof. Me. Jose Lucas Brandão Montes.

Trabalho de Conclusão de Curso (Superior de Tecnologia em Análise e Desenvolvimento de Sistemas) – Instituto Federal de Educação, Ciência e Tecnologia de Rondônia - IFRO, Vilhena, 2025.

1. Ensino. 2. IFRO. 3. Tecnologia. I. Montes, Jose Lucas Brandão (orient.). II. Instituto Federal de Educação, Ciência e Tecnologia de Rondônia - IFRO. III. Título.

Bibliotecário(a) Responsável: Rosilene Maria do Couto Marques, CRB-11/321

LUCAS FERREIRA RODRIGUES

Desenvolvimento da biblioteca Python AutoDBLoader

Artigo entregue como Trabalho de Conclusão de Curso apresentado ao Instituto Federal de Educação, Ciência e Tecnologia de Rondônia (IFRO), Campus Vilhena, como requisito parcial para obtenção do grau de Tecnólogo, junto ao Curso Superior em Tecnologia em Análise e Desenvolvimento de Sistemas, sob a orientação do professor José Lucas Brandão Montes.

Aprovado em 25/11/2025 pela banca examinadora.

Documento assinado digitalmente
gov.br
ERICK LEONARDO WEIL
Data: 04/12/2025 17:51:59-0300
Verifique em https://validar.iti.gov.br

_____

Prof. Especialista Erick Leonardo Weil

Examinador interno

Documento assinado digitalmente
gov.br
WESLEY JHONNES RAMOS ROLIM
Data: 02/12/2025 20:31:20-0300
Verifique em https://validar.iti.gov.br

_____ _____

Prof. Mestre Wesley Jhonnes Ramos Rolim

Examinador interno

Documento assinado digitalmente
gov.br
JOSE LUCAS BRANDAO MONTES
Data: 03/12/2025 15:28:52-0300
Verifique em https://validar.iti.gov.br

_____ _____

Prof. Mestre José Lucas Brandão Montes

Orientador

Resumo

O presente trabalho apresenta o desenvolvimento da biblioteca AutoDBLoader, criada para automatizar processos de extração, migração e inserção de dados em bancos de dados relacionais. Implementada em Python, a solução integra as bibliotecas Pandas e SQLAlchemy, oferecendo uma lógica de inserção ordenada a qual assegura a integridade referencial entre tabelas. Além da inserção e extração de dados, a biblioteca incorpora gerenciamento de memória, processamento em lote, registro de estado, logs detalhados e uma interface gráfica desenvolvida com PyPositron, facilitando sua configuração e uso. Os resultados demonstram que o AutoDBLoader constitui uma ferramenta robusta e escalável para migração e integração de dados entre múltiplos SGBDs

Palavras-chave: ensino; IFRO; tecnologia.

Abstract

This work presents the development of the AutoDBLoader library, created to automate the processes of extracting, migrating, and inserting data in relational databases. Implemented in Python, the solution integrates the Pandas and SQLAlchemy libraries, offering an orderly insertion logic that ensures referential integrity between tables. In addition to data insertion and extraction, the library incorporates memory management, batch processing, state tracking, detailed logging, and a graphical user interface developed with PyPositron, which facilitates its configuration and use. The results demonstrate that AutoDBLoader constitutes a robust and scalable tool for data migration and integration across multiple DBMSs.

Keywords: education, IFRO, technology.

1. Introdução

No cenário atual, é comum a necessidade de realizar processos de extração, migração e inserção de dados em bancos de dados relacionais. Seja para a criação de ambientes voltados à análise de tais informações quanto para a integração com novas ferramentas e funcionalidades (Carneiro; Moraes; Brayner, 2024). Nos ambientes analíticos, a utilização de arquivos semiestruturados no armazenamento e extração de dados vem sendo cada vez mais utilizados (Hamouda, 2024). Nesse contexto, a linguagem de programação Python destaca-se, sendo amplamente utilizada para solucionar diversos problemas analíticos, incluindo a construção de ambientes voltados à manipulação de dados em arquivos semiestruturados (Romano; Kruger, 2021). Essa popularidade surge, principalmente, devido ao seu ambiente rico em bibliotecas e pacotes capazes de solucionar problemas voltados a esses arquivos (Ramalho, 2022).

Entretanto, mesmo diante do vasto conjunto de bibliotecas e pacotes capazes de extrair e inserir dados em banco de dados relacionais, ainda não havia uma solução totalmente adequada para um problema, até a conclusão deste trabalho. O ponto central do meu trabalho encontra-se nesse seguinte problema, que é a dificuldade de inserir dados de arquivos semiestruturados em múltiplas tabelas de um banco de dados relacional, preocupando-se com relacionamento entre as tabelas e inserindo os registros na ordem correta, assim garantindo a integridade dos relacionamentos dos dados inseridos. Durante a idealização, deste trabalho, foi realizado o levantamento bibliográfico acerca do problema. Nesse caso, foram encontradas ferramentas que solucionam partes dele, porém não são adequadas para resolvê-lo por completo.

Em contrapartida, foram identificadas bibliotecas que solucionam partes desse problema, como o Pandas, amplamente utilizado na manipulação de dados tabulares semelhantes aos de bancos relacionais (Pandas Development Team, 2025). Do mesmo modo o SQLAlchemy, que possibilita a conexão a diferentes sistemas de gerenciamento de banco de dados (SGBDs) (SQLAlchemy Team, 2025). No entanto, essas bibliotecas, não solucionam por completo a questão abordada. Contudo, a junção de tais bibliotecas cria uma base para o início desse projeto.

Diante desse cenário, propõe-se o desenvolvimento da biblioteca AutoDBLoader, a qual integra funcionalidades do Pandas e do SQLAlchemy a outras bibliotecas, aliadas a uma lógica de programação própria. Dessa forma, torna-se possível extrair dados de um banco e inseri-los em outro sem a necessidade de intervenção manual na ordem de inserção ou no gerenciamento de relacionamentos entre tabelas ou na atualização de chaves primárias. Desse modo, transforma a biblioteca em uma solução viável para processos de migração de dados entre bancos relacionais. Além disso, possibilita a sua aplicação em outros casos de uso, como a construção de pipelines de ETL *(Extract, Transform, Load)*, devido a sua capacidade de realizar processos de extração e inserção de dados de forma independente.

Portanto, o propósito deste estudo é apresentar o desenvolvimento da biblioteca AutoDBLoader e demonstrar como ela soluciona o problema identificado por meio de uma lógica de inserção ordenada. Um processo que preserva a integridade dos relacionamentos entre os dados. Somando a isso, busca evidenciar como sua arquitetura automatizada simplifica processos complexos de extração e inserção de dados, reduzindo erros e aumentando a eficiência.

2. Fundamentação Teórica

2.1. Bancos de dados

Os bancos de dados surgiram na década de 1960, Em um momento que as empresas enfrentavam altos custos com o armazenamento manual e a indexação de arquivos. Nesse cenário, a IBM propôs o modelo hierárquico IMS (Information Management System), automatizando parte do processo, o que reduziu drasticamente os custos operacionais (Date, 2004, p. 70-79). devido a essa evolução tecnológica, os bancos de dados se consolidaram como parte essencial dos sistemas computacionais, permitindo operações de inserção, busca, alteração e exclusão de registros, possibilitando maior eficiência e organização da informação (Date, 2004; Elmasri; Navathe, 2011).

Com esse avanço surge modelo relacional, proposto por Ted Codd em 1970, o qual se tornou o paradigma dominante até os dias atuais (DB-Engines Ranking, 2025). Este modelo organiza os dados em tabelas compostas por linhas (tuplas) e colunas (atributos), onde a relação entre entidades é estabelecida por chaves primárias e estrangeiras (Elmasri; Navathe, 2011). Ademais, a integridade referencial garante que os relacionamentos entre tabelas permaneçam consistentes, evitando, assim, dados desconexos ou inválidos. (Elmasri; Navathe, 2011).

Diante desse cenário, para possibilitar a manipulação de dados nesse modelo, surge a linguagem SQL (Structured Query Language), padronizada e baseada em comandos de fácil compreensão, como SELECT, INSERT, UPDATE e DELETE (Date, 2004). Rapidamente se tornou a principal ferramenta para interação com bancos de dados, uma vez que, além de manipular dados, permite criar, modificar e excluir estruturas do banco, oferecendo ainda recursos como filtragem, agrupamento e junção de informações entre tabelas (Date, 2004).

Essas operações, por sua vez, são viabilizadas pelos Sistemas de Gerenciamento de Banco de Dados (SGBD), estes desempenham papel central ao administrar o acesso, a persistência e a estruturação das informações (Elmasri; Navathe, 2011, p.3-4). Entre esses sistemas, destacam-se MySQL, PostgreSQL, SQL Server e Oracle, os quais seguem princípios comuns como as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Tais propriedades são essenciais para assegurar a confiabilidade e a integridade das transações (Elmasri; Navathe, 2011, p.33-40).

A estrutura relacional, por sua vez, é organizada por meio dos relacionamentos entre tabelas, e eles podem ser de diferentes formas. No caso de relacionamentos um-para-um, registros de uma tabela correspondem a no máximo um registro em outra. Já nos relacionamentos um-para-muitos, um registro pode se associar a vários registros de outra tabela. Por fim, nos relacionamentos muitos-para-muitos, múltiplos registros de ambas as tabelas podem se interligar (Machado, 2018).

A partir dessa base estrutural, surge um desafio cada vez mais recorrente: a migração de dados. Esse processo envolve a transferência de informações entre diferentes ambientes, podendo incluir transformações na estrutura, tipos de dados ou restrições. Além disso, ele é fundamental para a modernização de sistemas legados, a integração com arquiteturas em nuvem e a adoção de soluções mais escaláveis e econômicas (Reis; Housley, 2022). No entanto, sua execução exige metodologias rigorosas para garantir a integridade e a consistência das informações migradas (Carneiro; Moraes; Brayner, 2024).

Nesse contexto, o processo de ETL (*Extract, transform, load*) torna-se indispensável. De forma prática, Ele consiste em extrair dados de diferentes fontes, transformá-los de acordo com regras de negócio, padronizações e requisitos analíticos, e carregá-los em um repositório final, como um data warehouse ou um data lake (Nwokeji; Matovu, 2021). Segundo Reis e Housley (2022), o ETL desempenha um papel central na engenharia de dados, pois não apenas movimenta informações, mas também garante sua qualidade, consistência e utilidade para consumo posterior. Dessa maneira, ele se estabelece como um componente essencial para a construção de sistemas de dados robustos e escaláveis, viabilizando análises confiáveis e processos de tomada de decisão baseados em dados.

2.2. Tipos de dados

De maneira geral, os dados manipulados nas pipeline ETL podem assumir três formas principais: estruturados, semi-estruturados e não estruturados. Especificamente, os dados estruturados são aqueles organizados de forma rígida em tabelas, com linhas e colunas definidas (Reis; Housley, 2022). O Parquet é um bom exemplo de arquivo com dados estruturados, de acordo com Vohra Reis e Housley (2022). Esse modelo colunar do Parquet reduz significativamente o espaço ocupado e melhora a performance de consultas analíticas, tornando-o especialmente adequado para ambientes de *big data*.

Por outro lado, os dados semi-estruturados possuem alguma organização, mas não seguem um esquema tão rígido quanto os estruturados. Como exemplos comuns, podemos citar documentos CSV, XML e JSON. Por isso, esses arquivos podem apresentar ou não hierarquias e relações implícitas entre os elementos, o que proporciona maior flexibilidade (Reis; Housley, 2022).

Por fim, os dados não estruturados são aqueles que não possuem um formato predefinido ou esquema fixo, como textos em documentos, páginas web ou PDFs. Apesar de serem abundantes e ricos em conteúdo, a falta de estrutura dificulta a sua utilização. Portanto, é necessário realizar diversas etapas de limpeza e transformação de dados para torná-los utilizáveis (Reis; Housley, 2022).

2.3. Python e bibliotecas

Com o crescimento do volume de dados, a linguagem Python se destaca como ferramenta fundamental para análise, manipulação e integração de sistemas voltados a dados (Ramalho, 2022). Desde sua criação ao final da década de 1980 por Guido van Rossum, o Python consolidou-se como a linguagem mais popular do mundo por sua simplicidade, versatilidade e vasta comunidade (Ramalho, 2022; Tiobe, 2025). Essa popularidade é fortalecida pelo PyPI (Python Package Index), repositório oficial que concentra centenas de milhares de bibliotecas, promovendo reutilização de código e acelerando o desenvolvimento de soluções (Romano; Kruger, 2021; PyPI, 2025).

Uma biblioteca de software pode ser entendida como um conjunto de funções e componentes reutilizáveis, que permitem padronização, confiabilidade e redução do esforço de desenvolvimento (Ramalho, 2022). Nesse sentido, o Pandas e o SQLAlchemy assumem papel de destaque. Enquanto o Pandas é amplamente utilizado para manipulação de dados tabulares, oferecendo operações de limpeza, transformação e integração com diversos formatos (Pandas Development Team, 2025). O SQLAlchemy por sua vez, facilita a interação com bancos relacionais, seja por meio de SQL direto ou via ORM, oferecendo compatibilidade com diferentes SGBDs e integração com o Pandas para extração de dados em formato de DataFrames (SQLAlchemy Team, 2025).

Assim, observa-se que a evolução dos bancos de dados, aliada ao desenvolvimento de ferramentas de manipulação e integração como SQL, SGBDs e bibliotecas Python, cria um ecossistema robusto e dinâmico. Esse ecossistema sustenta tanto aplicações tradicionais quanto modernas, que lidam com grandes volumes de dados e arquiteturas diversas. Além disso, ele fornece a base necessária para soluções como a proposta neste trabalho.

3. Materiais e métodos

O desenvolvimento da biblioteca AutoDBLoader foi conduzido com base no método ágil Kanban, escolhido por se tratar de um projeto individual que demandava flexibilidade e organização visual das tarefas, privilegiando a simplicidade e produtividade. Diferente de outras metodologias ágeis, como o Scrum, o Kanban não exige divisão do trabalho em sprints ou reuniões de acompanhamento, permitindo que as atividades fossem gerenciadas em fluxo contínuo, representado em colunas como “A Fazer”, “Em Progresso” e “Concluído”. Assim, o acompanhamento das etapas do

projeto tornou-se mais eficiente, facilitando a priorização das entregas e o gerenciamento das mudanças surgidas durante o desenvolvimento.

Além de organizar o fluxo de trabalho, o Kanban possibilitou a coleta de métricas relevantes de desempenho. Por exemplo, o Lead Time indica o tempo total entre a criação e a finalização de uma atividade, enquanto o Cycle Time mede o intervalo entre o início do desenvolvimento e sua conclusão. No contexto deste projeto, o Lead Time médio foi de 21,4 dias, e o Cycle Time médio alcançou 5,1 dias.

Referindo-se a linguagem de programação, o Python foi escolhido devido à sua ampla adoção na ciência e engenharia de dados e à grande variedade de bibliotecas voltadas à manipulação e integração de dados. Entre essas bibliotecas, destacam-se o Pandas, utilizado para manipulação de dados tabulares, e o SQLAlchemy, responsável pela conexão e abstração do acesso a múltiplos SGBDs, garantindo flexibilidade e robustez.

De forma complementar, no gerenciamento de versões, o Git foi utilizado em conjunto com a metodologia Gitflow, e o repositório hospedado no GitLab. Essa abordagem organizou o fluxo de desenvolvimento e permitiu integração com o GitLab CI/CD, automatizando o deploy das novas versões do AutoDBLoader. Complementando o processo, a documentação foi estruturada de forma clara, com docstrings no código e um README detalhado, orientando a instalação e o uso da biblioteca.

Além disso, para a construção da interface gráfica (GUI), a biblioteca PyPositron foi selecionada, permitindo a integração entre Python e JavaScript e garantindo a comunicação eficiente entre front-end e métodos da biblioteca. Nos testes e validação, o Cypress assegurou a confiabilidade da interface. Dessa forma, a metodologia adotada aliou a flexibilidade do Kanban à integração de ferramentas consolidadas, resultando em um processo incremental, estruturado e direcionado à solução de um problema relevante no contexto de migração e integração de dados.

4. Implementação

4.1. Evolução do escopo e ferramentas

No escopo inicial do projeto, a biblioteca contemplava apenas a funcionalidade de inserção de dados. Contudo, ao longo do desenvolvimento, surgiram novos requisitos, como a validação de parâmetros e dados de entrada, a geração de logs para acompanhamento das execuções, o registro dos estados de processamento e a necessidade de uma interface gráfica para configuração da ferramenta. Além disso, identificou-se a importância de também permitir a extração de dados de um banco de dados em formato compatível com a etapa de inserção. Dessa forma, essa

funcionalidade deveria ser capaz de extrair os registros de cada tabela e armazenar separadamente em um único arquivo nos formatos compatíveis com o método de inserção.

Devido a essa ampliação de escopo, a biblioteca passou a contemplar duas etapas do processo ETL convencional: extração e inserção de dados. Dessa forma, a AutoDBLoader tornou-se uma ferramenta de apoio ao processo ETL. Juntamente a isso, a capacidade de extrair, armazenar localmente e inserir em outro sistema garante suporte direto a processos de migração de bases de dados.

Ao analisarmos as versões iniciais do sistema ele utilizava a biblioteca mysql.connector, ela permitia apenas conexões com o MySQL e dependia exclusivamente de comandos SQL para manipulação dos dados, sem suporte a ORM (Object-Relational Mapper). Em razão dessas limitações, a solução foi substituída pela biblioteca SQLAlchemy, que oferece compatibilidade com múltiplos SGBDs, como MySQL, PostgreSQL, Oracle e SQLite, além de suporte a ORM.

Consequentemente, a mudança não apenas ampliou o número de bancos suportados, mas também trouxe maior abstração, permitindo a um mesmo código de inserção ser executado em diferentes SGBDs. Ademais, o SQLAlchemy viabilizou a visualização da estrutura do banco de dados, possibilitando identificar automaticamente chaves primárias e estrangeiras, colunas auto-incrementadas, relacionamentos entre tabelas e colunas com restrição de valores únicos.

Ao longo do desenvolvimento do projeto, o paradigma de programação passou por mudanças significativas. Inicialmente, o código era estruturado em funções monolíticas, o qual resultava em uma implementação complexa e de difícil compreensão. No entanto, com a migração da biblioteca mysql.connector para o SQLAlchemy, adotou-se um paradigma modular. Isso levou à divisão de grandes funções em blocos menores e reutilizáveis, resultando em maior clareza e melhor reaproveitamento do código.

Entretanto, mesmo com a modularização, as funções ainda permaneciam dispersas e apresentavam forte interdependência entre si. Diante desse cenário, o paradigma foi evoluído para a programação orientada a objetos (POO), passando a organizar as funções em classes e transformadas em métodos. Assim, a estrutura do sistema foi aprimorada, garantindo melhor organização, legibilidade e agrupamento de funções.

4.2. Arquitetura

A arquitetura da AutoDBLoader fundamenta-se na integração das bibliotecas Pandas e SQLAlchemy, cujas funções são complementares. Enquanto o SQLAlchemy estabelece

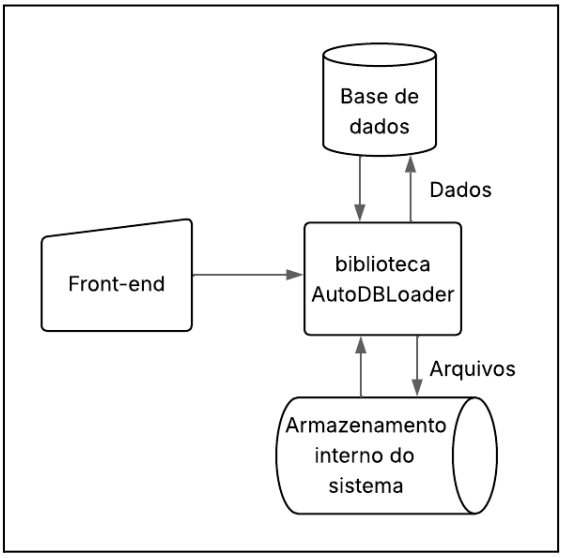
a conexão com o banco de dados e fornece informações sobre sua estrutura, o Pandas, por sua vez, atua na manipulação dos dados durante as etapas de extração e inserção.

No processo de extração, os registros são inicialmente lidos do banco e carregados na memória RAM no formato de DataFrame. Em seguida, esses dados são gravados na memória secundária em arquivos nos formatos CSV, Parquet ou JSON, funcionando como um repositório intermediário e independente do SGBD.

Em relação ao processo de inserção, o fluxo ocorre de forma inversa: os arquivos intermediários são carregados novamente para a memória RAM em formato de DataFrame e, a partir daí, inseridos nas tabelas do banco de destino.

Adicionalmente, a interface de usuário da AutoDBLoader é opcional e não constitui requisito para o funcionamento do sistema. Contudo, ela foi desenvolvida com o objetivo de simplificar a configuração e o uso da biblioteca. Ela permite ao usuário configurar visualmente a conexão com o banco e as tabelas. Além de gerar automaticamente o arquivo JSON de configuração necessário e possibilitar a execução diretamente pela interface. Dessa forma, entra como uma peça fundamental que facilita a utilização da AutoDBLoader. Na figura abaixo, há a ilustração da arquitetura implementada:

Figura 1. Arquitetura da biblioteca AutoDBLoader



Fonte: Autoria própria (2025)

Na imagem acima é apresentado de forma visual a interação entre o *Front-end*, a biblioteca AutoDBLoader, o armazenamento interno e a base de dados. A biblioteca realiza a leitura de arquivos e a inserção dos dados no banco de dados ou a biblioteca extrai os dados do banco e os salva em arquivos no armazenamento interno.

4.3. Diagrama de classes

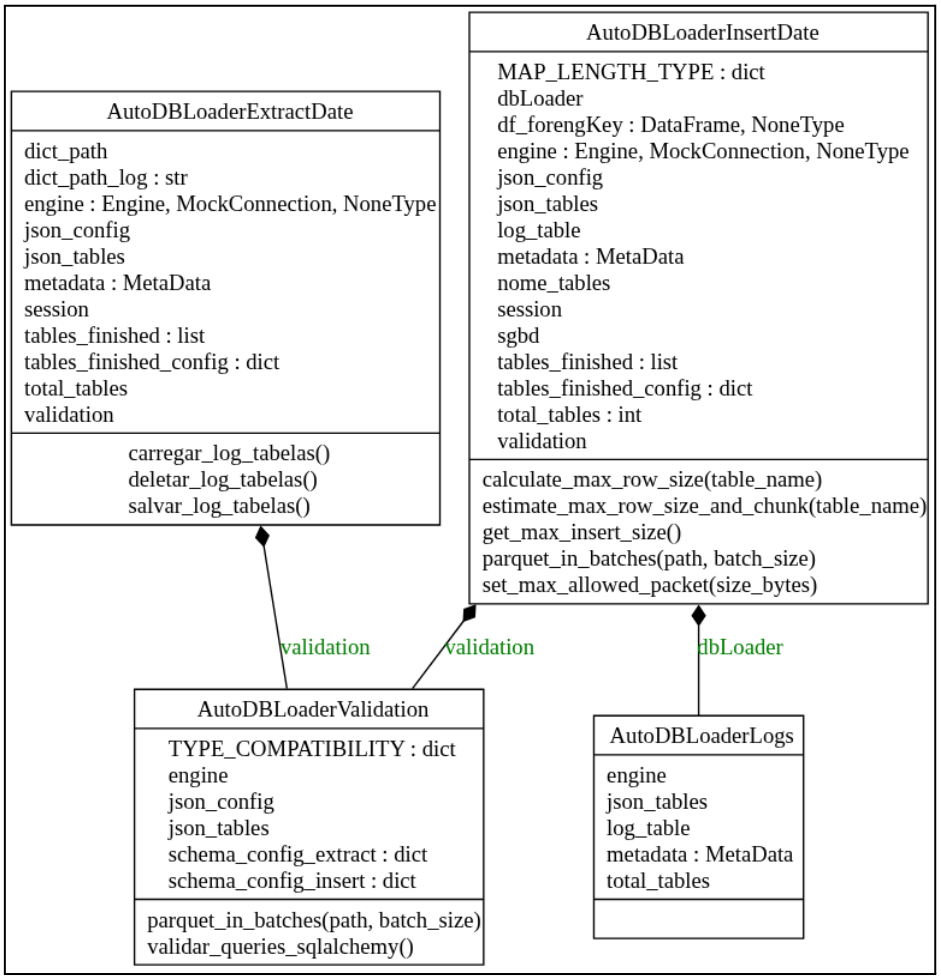
O paradigma adotado para o desenvolvimento da biblioteca foi a Programação Orientada a Objetos (POO). Nesse paradigma, funções que desempenham papéis semelhantes ou possuem correlação são agrupadas em classes, e podem ser instanciadas por outras classes, estabelecendo assim os relacionamentos entre elas.

O projeto conta com quatro classes principais:

- AutoDBLoaderValidation: responsável por validar as configurações de entrada fornecidas pelo usuário, bem como as configurações do banco de dados, garantindo que os dados estejam corretos antes da execução dos processos.
- AutoDBLoaderSystemLogs: gerencia o armazenamento, a extração e a exclusão de registros relacionados ao estado do processamento, mantendo um histórico de execução no banco de dados.
- AutoDBLoaderExtractDB: realiza o processo de extração de dados e, durante sua execução, instancia a classe AutoDBLoaderValidation para utilizar seus métodos de validação.
- AutoDBLoaderInsertLoader: encarregada da inserção dos dados no banco de dados, essa classe instância tanto a AutoDBLoaderValidation, para assegurar a consistência das informações, quanto a AutoDBLoaderSystemLogs, para registrar o estado do processamento, em caso de erro.

Abaixo consta a figura 2 que contém o diagrama de classes da AutoDBLoader:

Figura 2. Arquitetura da biblioteca AutoDBLoader



Fonte: Autoria própria (2025)

O diagrama acima ilustra as classes da biblioteca AutoDBLoader representando a arquitetura interna da ferramenta. Ele mostra as classes e suas interações, destacando os módulos de leitura, tratamento e inserção de dados. Por fim, demonstra a abordagem orientada a objetos, que garante clareza e fácil manutenção do código.

Dessa forma, o diagrama de classes evidencia a interação e a dependência entre as classes. Assim, mostrando como cada componente da biblioteca se relaciona para garantir a integridade dos dados e o registro do processo de execução.

4.4. Funcionalidade de inserção de dados

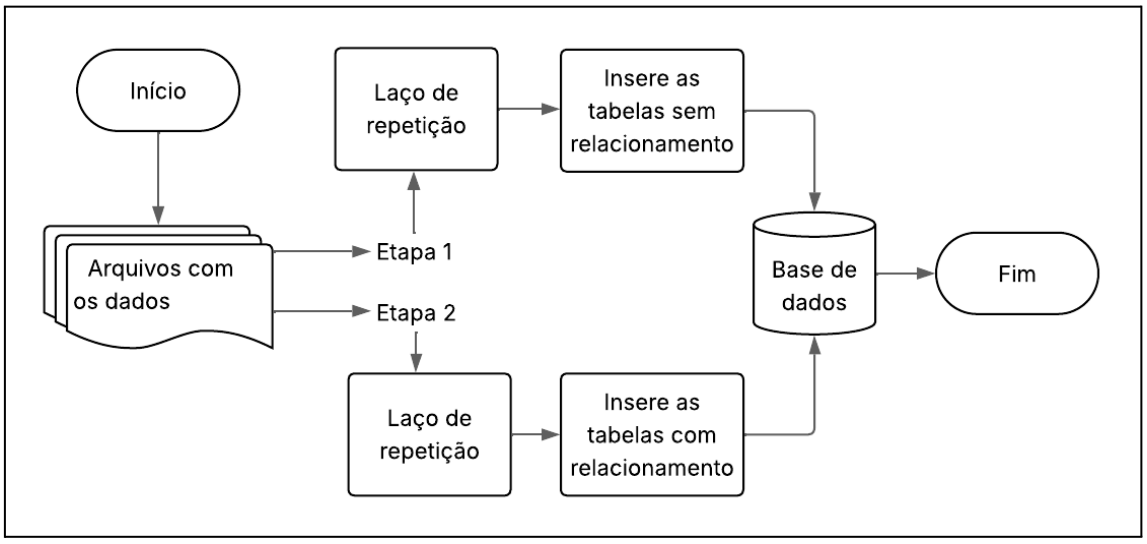
Sendo a principal funcionalidade da biblioteca, o processo de inserção de dados originou o desenvolvimento deste projeto que visa resolver a lógica de inserção dos dados em banco de dados relacionais. Nesse contexto, a complexidade desse processo está nos relacionamentos entre as tabelas: em muitos casos, uma tabela armazena referências a registros de outra, representando a ligação entre seus dados.

Durante a inserção, essa lógica pode ser comprometida caso o valor de uma chave primária seja alterado e a atualização não seja refletida nas tabelas relacionadas. Como resultado, isso pode ocasionar perda de integridade referencial ao gerar informações inconsistentes ou até falhas no processo de inserção, especialmente ao tentar inserir referências a registros ainda não existentes.

Diante dessa problemática, tornou-se necessário desenvolver um algoritmo capaz de organizar a ordem de inserção dos dados. Inicialmente, o algoritmo prioriza a inserção das tabelas que não possuem chaves estrangeiras e, posteriormente, processa aquelas que dependem de relacionamentos. Para isso, a estrutura do banco deve estar previamente criada, permitindo ao algoritmo verificar se as tabelas referenciadas já foram carregadas e ajuste as chaves estrangeiras de acordo com os novos valores atribuídos às chaves primárias. Dessa forma, garante-se a integridade referencial em todo o processo.

Na figura abaixo, há o fluxograma ilustrativo da lógica de inserção na visão macro.

Figura 3. Arquitetura da biblioteca AutoDBLoader



Fonte: Autoria própria (2025)

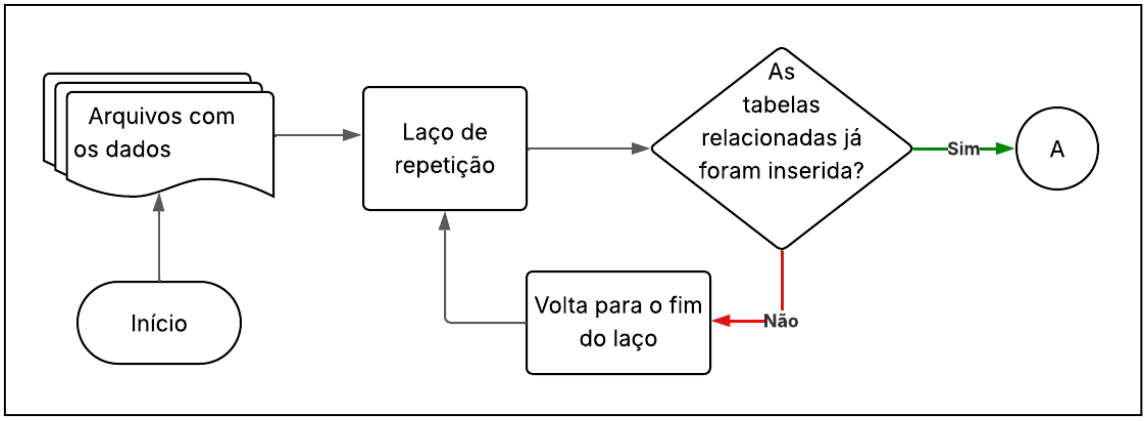
A figura acima representa o fluxo geral de inserção de dados na AutoDBLoader. O diagrama mostra o processo completo de inserção de dados, dividido em duas etapas: a primeira insere as tabelas sem relacionamento, e a segunda insere as tabelas relacionadas, garantindo a integridade dos dados na base.

De forma detalhada, o algoritmo classifica as tabelas em dois grupos: aquelas que possuem chaves estrangeiras e as que não possuem. Primeiramente, o processo de inserção é realizado pelas tabelas sem dependências, pois estas não necessitam de

referências a registros prévios. Antes da inserção, o sistema adiciona temporariamente uma coluna denominada *old_id_insert*, destinada a armazenar o identificador original de cada registro no banco de origem. Em seguida, os dados dessas tabelas são carregados no novo banco, e cada tabela é marcada como concluída.

Em seguida, inicia-se o laço de inserção das tabelas onde contêm chaves estrangeiras. Pela lógica uma tabela dependente só é processada, se todas as tabelas referenciadas por ela estiverem previamente preenchidas. Caso contrário, a tabela retorna ao final do laço, enquanto o algoritmo verifica outra tabela disponível para inserção. Dessa maneira, o processo garante que todas as referências estejam consistentes antes da inserção, assegurando a integridade referencial. A Figura 4 a seguir ilustra esse fluxo de forma esquemática.

Figura 4. Parte 1 do fluxograma de inserção dos dados da biblioteca AutoDBLoader



Fonte: Autoria própria (2025)

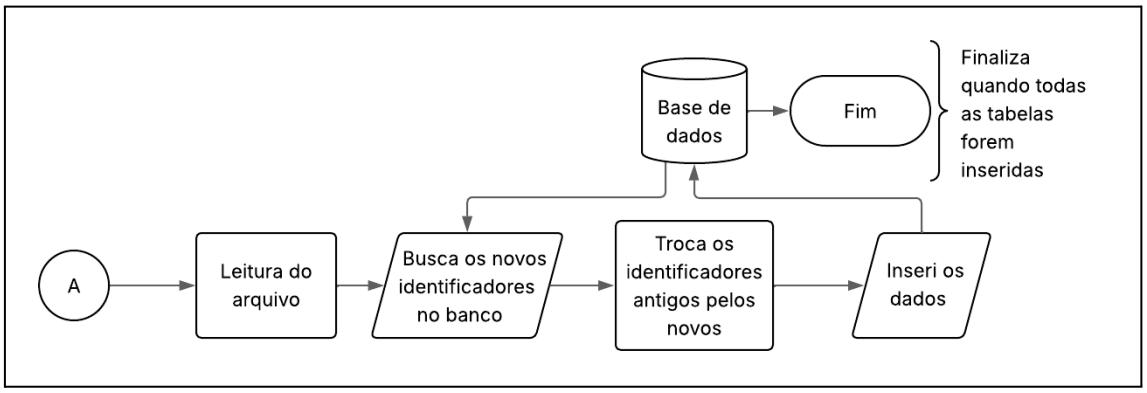
A figura acima contém a parte 1 do fluxograma da AutoDBLoader, ela descreve o processo inicial de verificação e inserção das tabelas relacionadas. Nessa fase, a execução segue um laço de repetição que percorre os arquivos de dados, assegurando, de maneira ordenada e consistente, que todas as tabelas dependentes sejam devidamente inseridas.

Posteriormente, são utilizados os valores armazenados na coluna *old_id_insert*, desse modo as antigas chaves estrangeiras sejam substituídas pelas novas chaves primárias geradas durante o processo de inserção. Essa atualização assegura os relacionamentos entre os dados, deixando-os inalterados, mantendo as referências correspondentes aos identificadores originais.

Para cada lote, o sistema executa uma query no banco e ele retorna apenas os registros daquele intervalo, considerando somente duas colunas: o novo identificador e o *old_id_insert* correspondente. Em seguida, é realizado um *merge* no Pandas entre os dados a serem inseridos e o resultado da query, utilizando o ID antigo como chave de

junção. Dessa forma, as chaves estrangeiras são substituídas pelos novos identificadores de maneira eficiente, garantindo a integridade dos dados. A Figura 5 a seguir ilustra esse processo de forma visual.

Figura 5. Parte 2 do fluxograma de inserção dos dados da biblioteca AutoDBLoader



Fonte: Autoria própria (2025)

A figura acima contém a parte 2 do fluxograma, ela apresenta em detalhe o processo de leitura e inserção dos dados no banco de dados. Nessa fase, a AutoDBLoader realiza a leitura dos arquivos, identifica e substitui os antigos identificadores pelos novos e, em seguida, efetua a inserção dos registros. O processo é finalizado quando todas as tabelas relacionadas são devidamente inseridas.

Com tudo, o algoritmo realiza processos adicionais, como a verificação de colunas com valores únicos, bem como o tratamento de chaves primárias não auto-incrementadas. Nessas situações, a coluna *old_id_insert* não é criada, e os identificadores originais do banco de origem são mantidos. Por fim, ao término de todo o procedimento, a coluna *old_id_insert* é removida de todas as tabelas, garantindo que o banco de destino permaneça consistente e íntegro.

4.5. Gerenciamento de memória e processamento em lote

Durante o desenvolvimento do projeto, identificou-se um problema crítico ao tentar inserir 1 GB de dados em um único teste, pois o sistema consumiu toda a memória RAM disponível, resultando em um crash. Essa situação ocorreu devido ao carregamento integral de arquivos muito grandes na memória, excedendo a capacidade do sistema. Diante dessa limitação, foi necessário implementar um mecanismo de gerenciamento de memória, utilizando o método *virtual_memory* da biblioteca Python *psutil*. Por meio desse recurso, a quantidade de RAM livre no sistema pode ser monitorada, permitindo a utilização, de forma controlada. Foi destinado até 60% desse espaço para a leitura dos arquivos, garantindo maior eficiência e estabilidade do processo.

Além disso, esse gerenciamento foi combinado com o processo de leitura em chunks. Nesse mecanismo, o sistema calcula o tamanho aproximado em bytes de cada registro do arquivo, considerando os tipos das colunas. A partir desse cálculo, divide o espaço de memória disponível pelo tamanho estimado de cada registro, obtendo o número máximo de linhas capazes de ser processadas por chunk. Dessa forma, o sistema evita estouros de memória ao dividir arquivos grandes em partes menores. Ao mesmo tempo, consegue ler o maior volume possível de dados por vez, resultando em uma melhor performance de inserção.

Porém, não basta apenas controlar a memória disponível do computador; também é fundamental gerenciar o tamanho máximo de comandos INSERT aceito pelo SGBD. Para atender a essa exigência, realiza-se uma query no banco onde ele retorna o valor da variável *max_allowed_packet*, a qual indica o tamanho máximo permitido para uma query. Com essa informação, e considerando o tamanho médio de cada registro do arquivo, torna-se possível calcular a quantidade máxima de registros cabíveis de serem inseridos de uma só vez sem exceder o limite do SGBD.

Portanto, além de dividir o arquivo em chunks de leitura, cada chunk é subdividido em blocos menores para inserção no banco. Esse procedimento garante ao sistema inserir o maior número de linhas possível por vez, preservando a performance e evitando erros relacionados ao estouro do limite. Ademais, essa abordagem integra o gerenciamento de memória com o controle do tamanho das queries, promovendo maior eficiência e confiabilidade no processo de inserção de dados.

4.6. Funcionalidade de extração de dados

Embora seja uma funcionalidade secundária da aplicação, o processo de extração de dados foi incorporado com o objetivo de simplificar e automatizar essa etapa. Para sua implementação, o sistema utiliza a biblioteca SQLAlchemy para estabelecer a conexão com o banco de dados e, em seguida, o método sql_query da biblioteca Pandas para realizar a extração personalizada dos registros. Nesse fluxo, os dados de cada tabela são exportados para um único arquivo, que pode ser gerado nos formatos CSV, JSON ou Parquet, de acordo com a necessidade do usuário.

Adicionalmente, o sistema oferece suporte à execução de consultas personalizadas, ampliando a flexibilidade da extração. Esse processo é acompanhado por um mecanismo de validação robusto. Sendo responsável por verificar o JSON de configuração, a conexão com o banco de dados, o diretório de destino, a existência das tabelas especificadas e a consistência das consultas fornecidas. Por fim, em situações de falha, o sistema é capaz de registrar o estado parcial da extração, permitindo o retorno da execução exatamente do ponto onde foi interrompida.

4.7. Sistema de registro de estado do processamento

Durante o processo de inserção de dados, é comum a ocorrência de erros, interrompendo assim a execução. Para evitar a perda total do progresso, foi criado um sistema de registro de estado do processamento, responsável por armazenar as tabelas já finalizadas. Essas informações são salvas diretamente no banco de dados em uma tabela auxiliar chamada autodbloader_log, ela contém uma string com os nomes das tabelas concluídas, separadas por vírgulas.

Ao iniciar um novo processo de inserção, o sistema verifica a existência da tabela *autodbloader_log*. Caso ela esteja presente, as tabelas nela registradas são adicionadas à lista de tabelas já finalizadas, permitindo que o processamento seja retomado a partir desse ponto. Concluída a execução, a tabela é excluída. Porém se a tabela não existir, o sistema segue o fluxo padrão. Dessa forma, mesmo que o usuário inicie o processo em uma máquina e termine em outra, o processo já realizado não é perdido, pois o histórico fica salvo no banco.

De modo semelhante, o processo de extração também possui um controle de progresso. Nesse caso, é criado um arquivo JSON que registra as tabelas já processadas, permitindo que, em caso de falha, o sistema retome a execução de forma contínua e sem duplicidades. Quando a extração é concluída com sucesso, o arquivo é removido, assegurando assim a limpeza e o bom funcionamento do ambiente de trabalho.

4.8. Sistema de logs de acompanhamento

A AutoDBLoader conta com um sistema de logs o qual permite o monitoramento completo da execução, tanto nas etapas de extração quanto nas de inserção de dados. Para sua implementação, foi empregada a biblioteca Logging, amplamente utilizada em aplicações Python pela eficiência no registro e na gestão de eventos.

Os logs seguem uma padronização cromática facilitando a interpretação dos resultados: o verde indica sucesso nas operações, o amarelo sinaliza alertas o qual requerem atenção e o vermelho destaca erros que exigem intervenção. Com isso, o usuário consegue identificar rapidamente o estado do processo, sem depender de descrições extensas.

Durante a execução, são registradas informações detalhadas sobre cada etapa, abrangendo desde as validações iniciais até os procedimentos internos. No método de extração, os logs indicam as tabelas já processadas, enquanto, no método de inserção, informam a tabela e o lote em execução. Esse recurso garante maior transparência, facilita a detecção de falhas e contribui para o acompanhamento preciso do desempenho da biblioteca.

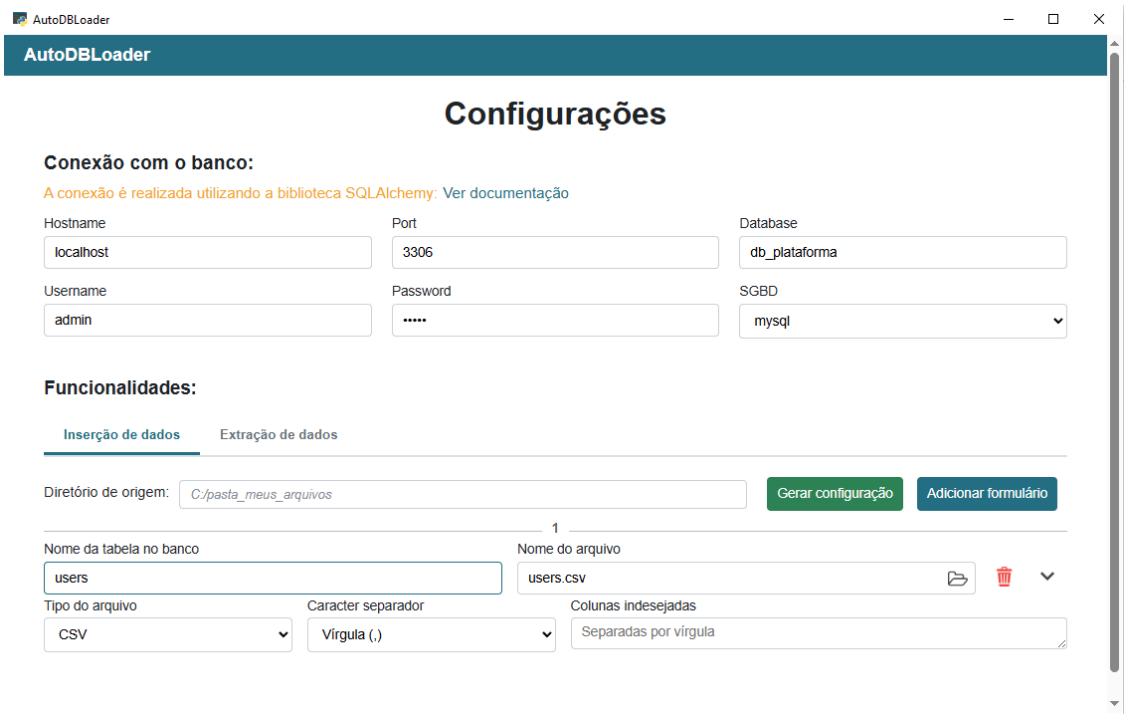
4.9. Interface de usuário

A interface de usuário foi desenvolvida com o objetivo de simplificar a configuração da biblioteca, oferecendo uma alternativa visual e intuitiva ao processo. Embora não seja

fundamental para a sua utilização, ela representa um recurso complementar auxiliando na definição das conexões com o banco de dados e a seleção das tabelas que participarão do processo, seja de inserção ou de extração de dados.

A instalação da interface ocorre juntamente com a biblioteca, além de sua inicialização ser bastante simples, sendo necessário apenas abrir um terminal e executar o comando "autodbloader". A interface será carregada imediatamente e, ao ser encerrada, o histórico de configuração é descartado. Uma vez iniciada, a interface apresenta a tela inicial, exibida na Figura 6.

Figura 6. Tela inicial da interface de configuração da biblioteca AutoDBLoader



Fonte: Autoria própria (2025)

A figura acima apresenta a tela inicial da interface de configuração da AutoDBLoader. Nela são exibidos os campos para preenchimento dos parâmetros de conexão, como as informações do banco de dados, além dos parâmetros de configuração específicos de cada tabela.

A partir dela, o usuário pode configurar manualmente os parâmetros necessários e, em seguida, gerar automaticamente um arquivo JSON de configuração, que será utilizado para executar os processos da biblioteca.

Esse arquivo pode ser empregado tanto em um script Python independente quanto diretamente na própria interface, a qual se comunica com os métodos internos da

biblioteca para realizar as operações configuradas. Durante a execução, os *logs* são exibidos no terminal que originou a inicialização da interface, permitindo o acompanhamento em tempo real. A Figura 7 apresenta a tela correspondente à etapa de execução.

Figura 7. Tela execução dos métodos da biblioteca AutoDBLoader

Fonte: Autoria própria (2025)

A figura acima apresenta a tela de execução dos métodos da biblioteca. Nela é exibido o JSON de configuração, que pode ser copiado e utilizado em um arquivo Python para execução direta. Além disso, o usuário tem a opção de executar o método diretamente pela interface, clicando no botão “Executar”.

O desenvolvimento da interface foi conduzido por meio da biblioteca PyPositron, um framework que possibilita a criação de aplicações desktop baseadas em HTML, CSS e JavaScript. Essa escolha mostrou-se estratégica, pois viabilizou a integração eficiente entre a lógica de programação em Python e os recursos de interface oferecidos pelo JavaScript, garantindo, assim, a comunicação direta entre ambos. Dessa forma, tornou-se possível executar métodos Python a partir de funções JavaScript, o qual proporcionou maior fluidez e dinamismo à interação entre as camadas do sistema.

Quanto ao gerenciamento de estados, foi necessário desenvolver uma lógica própria, uma vez que a interface é estruturada principalmente por formulários de configuração. Nesse cenário, adotou-se o local storage como mecanismo para armazenar arquivos JSON contendo as informações inseridas pelo usuário. Desse modo, quando o usuário retorna a uma tela previamente acessada, o sistema é capaz de restaurar automaticamente o estado anterior, promovendo continuidade e praticidade na navegação.

Por fim, o mesmo mecanismo é empregado na geração do arquivo de configuração da biblioteca. Reunindo as informações de todos os formulários em uma única estrutura JSON. Sendo plenamente compatível com os métodos da AutoDBLoader, assegurando coerência e padronização em todo o processo.

5. Resultados

5.1. Ferramenta Entregue

Ao término do desenvolvimento, a biblioteca AutoDBLoader consolidou-se como uma ferramenta robusta e funcional, capaz de realizar processos automáticos de extração e inserção de dados em bancos relacionais. Para isso, utiliza arquivos nos formatos CSV, JSON e Parquet como meio de armazenamento intermediário. A solução garante a integridade referencial dos registros, além de incorporar sistemas de validação, logs de acompanhamento e registro de estado do processamento, que asseguram confiabilidade e rastreabilidade.

A primeira versão estável foi publicada oficialmente no gerenciador de pacotes PyPI em 13 de agosto de 2025, tornando-a acessível à comunidade Python. Juntamente com o pacote, foi disponibilizada uma documentação detalhada e uma interface gráfica de configuração. Desenvolvida para facilitar o uso por usuários sem experiência prévia em programação.

5.2. Solução para o Problema Identificado

O problema central abordado refere-se à dificuldade de realizar processos de extração e inserção de dados em bancos relacionais a partir de arquivos estruturados e semiestruturados. Nesse contexto, a AutoDBLoader surge como uma solução eficiente ao automatizar etapas complexas desses processos.

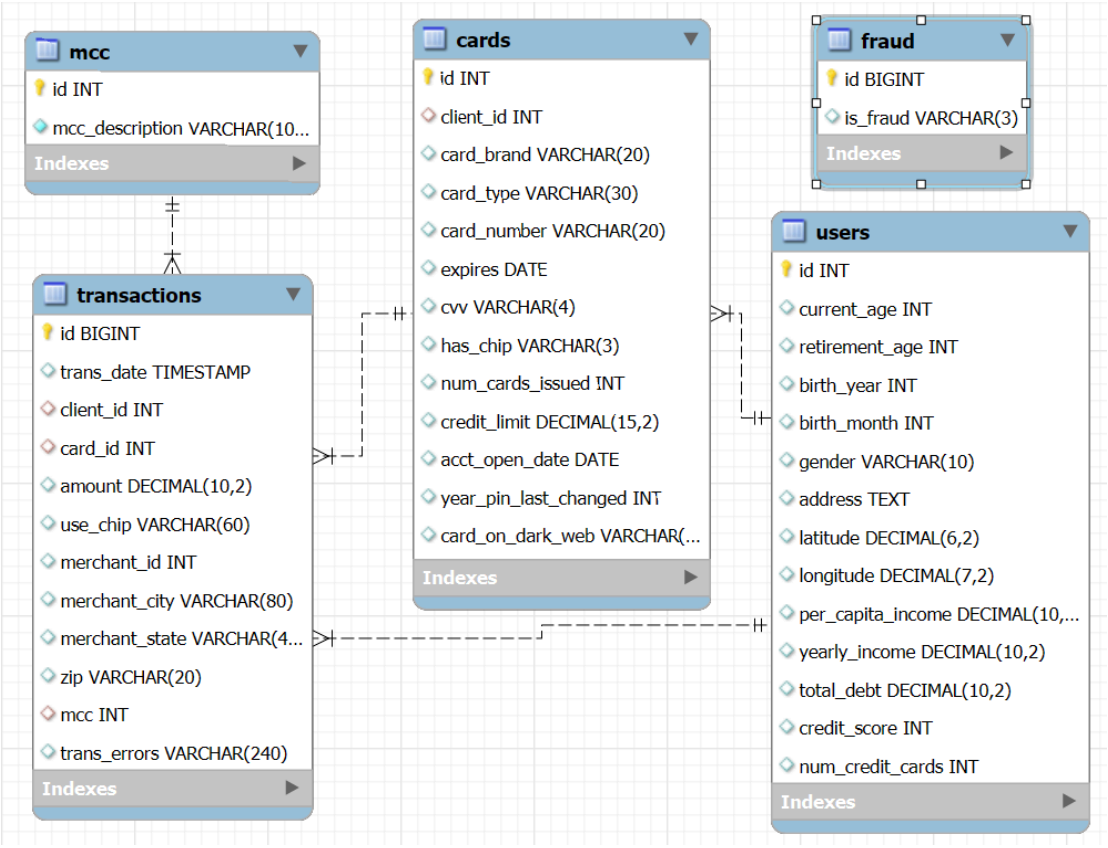
Na extração de dados, a biblioteca permite obter informações de forma rápida e configurável, sem necessidade de código SQL. Já na inserção, executa todas as etapas de maneira automatizada, garantindo a preservação das chaves estrangeiras e a integridade dos dados. Diante disso, a combinação desses dois processos, gera uma solução completa e com amplas utilidades.

A ferramenta também dispõe de uma interface gráfica intuitiva, que possibilita configurar e executar os processos de forma visual. Ademais, a AutoDBLoader pode ser empregada na construção de pipelines de ETL (Extract, Transform, Load). Devido a realizar as etapas de extração e inserção, enquanto a transformação pode ser realizada externamente. Assim, ampliando sua aplicabilidade em diferentes cenários de migração e integração de dados.

5.3. Testes de performance

Para a execução dos testes de performance da biblioteca AutoDBLoader, foi utilizado o banco de dados público transactions-fraud-datasets, disponível na plataforma *Kaggle*. Esse conjunto de dados contém informações de transações financeiras, classificadas como verídicas ou fraudulentas. Totalizando mais de 18 milhões de registros e aproximadamente 923 MB de dados armazenados em arquivos no formato CSV. Essa escolha se deu por se tratar de um dataset robusto e amplamente utilizado em pesquisas relacionadas à análise de dados e detecção de fraudes.

Figura 8. Modelagem do banco de dados utilizado no teste



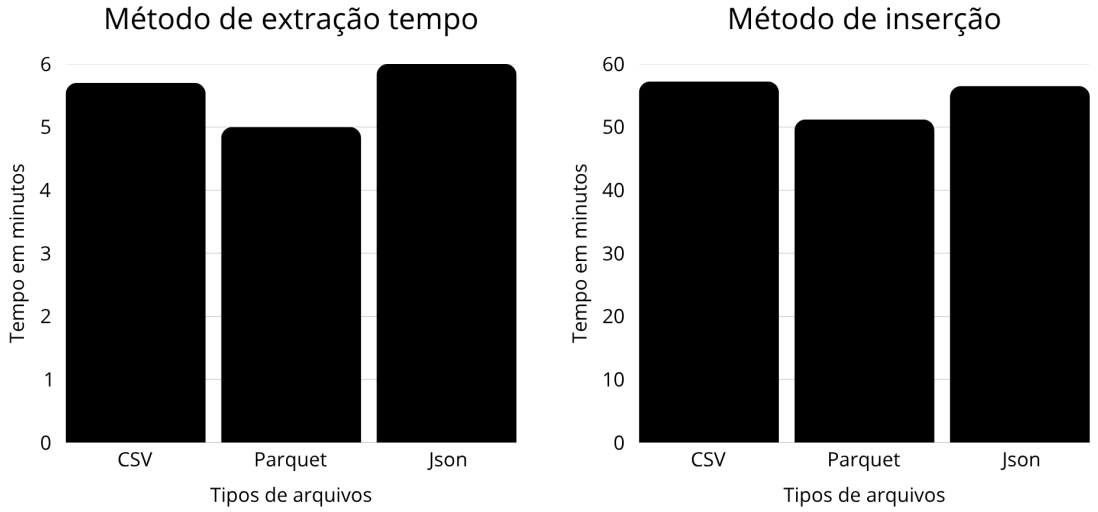
Fonte: Autoria própria (2025)

A figura acima apresenta a modelagem do banco de dados utilizado nos testes. O banco é composto por cinco tabelas: Transactions, Users, Fraud, MCC e Cards. Todas as tabelas possuem relacionamentos entre si, com exceção da Fraud, e totalizando 44 atributos no conjunto.

Em relação ao ambiente de testes, o experimento foi conduzido em um computador com 16GB de memória DDR4, processador AMD Ryzen 7 2700 e SSD NVMe de 256 GB. O sistema gerenciador de banco de dados (SGBD) utilizado foi o MySQL, executado localmente por meio do *MySQL Server*. Além disso, o código SQL para criação das tabelas e estrutura do banco foi elaborado manualmente. É importante destacar que, devido à execução simultânea do SGBD e da biblioteca na mesma máquina, houve compartilhamento dos recursos computacionais disponíveis, o qual impacta diretamente no desempenho global do processo.

Os resultados obtidos demonstraram que a biblioteca apresentou desempenho consistente nas diferentes modalidades de teste. No processo de inserção de dados, a AutoDBLoader inseriu os 18 milhões de registros distribuídos em cinco tabelas em 57 minutos e 15 segundos utilizando arquivos CSV. Quando utilizados arquivos Parquet, o tempo de inserção foi reduzido para 51 minutos e 12 segundos, enquanto o uso de JSON resultou em 56 minutos e 34 segundos. Já no processo de extração de dados, o tempo necessário para exportação em formato CSV foi de 5 minutos e 42 segundos, em Parquet de 4 minutos e 56 segundos, e em JSON de 5 minutos e 54 segundos.

Figura 9. Gráfico das métricas alcançadas nos testes



Fonte: Autoria própria (2025)

A figura acima contém dois gráficos, neles são apresentados os tempos de execução dos dois métodos da biblioteca. Com esses gráficos é possível identificar visualmente a diferença no tempo de execução entre os 3 tipos de arquivos suportados, CSV, Parquet e Json. Sendo possível perceber que o Parquet possui a melhor performance nos dois métodos.

Dessa forma, conclui-se que o formato Parquet mostrou-se o mais adequado para os processos de inserção e extração de dados na AutoDBLoader, por apresentar o melhor desempenho em tempo de execução e menor ocupação de espaço em disco. Assim, seu uso é recomendado para cenários onde envolvam grandes volumes de dados e demandam alta eficiência no processamento.

5.4. Antes e depois da autoDBLoader

Antes do desenvolvimento da AutoDBLoader, a inserção de dados em bancos relacionais com bibliotecas do ecossistema Python exigia diversas etapas manuais. Era necessário compreender o funcionamento de ferramentas como Pandas e SQLAlchemy. E também criar scripts personalizados para ler arquivos, converter dados em instruções SQL e enviá-las ao banco, o que tornava o processo lento e propenso a erros.

Além disso, tabelas com chaves estrangeiras exigiam a atualização manual das novas chaves geradas. juntamente a isso o gerenciamento de memória e o limite de tamanho das instruções SQL representavam desafios adicionais, especialmente com grandes volumes de dados. Diferenças de tipos, colunas ou valores ausentes também demandavam verificações e tratamentos extras.

A AutoDBLoader surgiu para eliminar essas limitações, automatizando todo o fluxo de extração, conversão e inserção de dados. Com ela, o processo se torna ordenado, seguro e eficiente. Ela mantém a integridade referencial e reduz drasticamente o esforço técnico necessário para lidar com grandes cargas de dados.

5.5. Comparação com outras ferramentas

Durante o desenvolvimento da AutoDBLoader, foi realizada uma análise comparativa com outras ferramentas que possuem objetivos semelhantes. Especialmente no que se refere à extração e inserção de dados em bancos relacionais a partir de arquivos estruturados e semiestruturados. Entre as soluções avaliadas destacam-se o Dask, o Petl e o Apache NiFi, amplamente utilizados em processos de integração e transformação de dados.

O Dask é uma biblioteca Python voltada para o processamento paralelo e distribuído de grandes volumes de dados. Embora ofereça desempenho elevado e suporte aos formatos CSV, JSON e Parquet, sua utilização requer configurações adicionais e conhecimentos técnicos avançados sobre paralelismo e gerenciamento de clusters. Além disso, o Dask não provê mecanismos nativos para manter a integridade

referencial entre tabelas. Sendo mais adequado para processamento analítico do que para inserção estruturada em bancos de dados relacionais.

O Petl, por sua vez, é uma biblioteca Python voltada para operações de *Extract, Transform, Load (ETL)* em pequena escala. Ele permite a leitura de arquivos e a inserção de registros em bancos de dados. Porém não oferece recursos automatizados para controle de dependências entre tabelas, validação de dados ou gerenciamento de memória durante a carga. Dessa forma, apesar de simples e eficiente em contextos limitados, o Petl se mostra insuficiente para cenários de migração complexa que envolvem múltiplas relações entre tabelas e grandes volumes de informação.

Já o Apache NiFi é uma plataforma robusta de integração de dados a qual se destaca por sua interface visual, permitindo o desenho de fluxos de ETL por meio de componentes gráficos. Apesar de sua flexibilidade e alto desempenho, a ferramenta apresenta uma curva de aprendizado acentuada, requer um ambiente Java configurado e é mais adequada para contextos corporativos de grande porte. Além disso, a complexidade de sua instalação e manutenção a torna menos acessível para desenvolvedores que buscam soluções leves e integradas ao ecossistema Python.

Em contraste, a AutoDBLoader propõe uma abordagem mais direta e acessível. Desenvolvida inteiramente em Python, ela automatiza de forma completa o processo de extração e inserção de dados, garantindo a integridade referencial dos registros e o controle de memória durante a execução. Diferentemente das ferramentas mencionadas, não exige configurações complexas nem conhecimento avançado de infraestrutura. E conta com uma interface gráfica permitindo aos usuários configurarem e executarem operações de forma intuitiva, tornando o processo de migração e carregamento de dados mais ágil, confiável e acessível.

Em síntese, enquanto o Dask e o Petl se concentram em aspectos específicos do processamento de dados e o Apache NiFi em soluções empresariais de larga escala, a AutoDBLoader se destaca por oferecer uma solução unificada, leve e automatizada para extração e inserção de dados em bancos relacionais, conciliando praticidade, desempenho e robustez dentro de um único ambiente.

5.6. Limitações e novas funcionalidades

Apesar da AutoDBLoader ter sido desenvolvida com o apoio de bibliotecas ágeis e eficientes. Ela tem sua performance limitada pelos recursos da máquina em que está sendo executada, bem como pelo tamanho máximo de instruções *INSERT* aceito pelo SGBD e pelas próprias restrições de inserção do Pandas. Ainda assim, em contextos que envolvem *datasets* de pequeno e médio porte, a biblioteca apresenta um desempenho satisfatório, realizando migrações de forma estável e com boa velocidade.

Como evolução futura, prevê-se o desenvolvimento de uma funcionalidade de atualização incremental de dados. Onde permitirá identificar e inserir apenas os registros inexistentes no banco e atualizar automaticamente os registros modificados. Essa melhoria possibilitará o uso da AutoDBLoader não apenas para migrações pontuais, mas também para a atualização contínua de ambientes analíticos, como *data warehouses*. Dessa forma, ampliando seu potencial de aplicação no ecossistema de engenharia e integração de dados.

6. Conclusão

O desenvolvimento da biblioteca AutoDBLoader representou uma solução prática e inovadora para os desafios de migração, extração e inserção de dados em bancos relacionais. Desde o início, o projeto foi idealizado para reduzir a complexidade e o tempo de execução dessas tarefas. Automatizando processos que tradicionalmente exigem grande esforço manual e conhecimento técnico avançado. Dessa forma, a AutoDBLoader consolida-se como uma ferramenta onde une simplicidade de uso, robustez técnica e eficiência computacional.

Durante o desenvolvimento, adotou-se uma abordagem orientada a objetos, garantindo modularidade, reutilização e facilidade de manutenção. O uso de bibliotecas consolidadas, como Pandas e SQLAlchemy, proporcionou uma base segura e flexível para manipulação e persistência de dados. Assim, a AutoDBLoader tornou-se capaz de lidar com diferentes formatos CSV, JSON e Parquet, adaptando-se a diversos cenários de uso e necessidades de integração.

Nos testes de desempenho, realizados com um dataset público de 18 milhões de registros, a biblioteca demonstrou eficiência e estabilidade, realizando inserções e extrações de grandes volumes de dados com controle de memória e gerenciamento dinâmico de lotes. Entre os formatos avaliados, o Parquet apresentou o melhor resultado, concluindo o processo de inserção em aproximadamente 50 minutos e utilizando menos espaço em disco, seguido por CSV e JSON, com tempos ligeiramente maiores.

O sistema de logs detalhados e o controle automático de memória reforçaram a confiabilidade e a transparência da ferramenta. Os resultados obtidos evidenciam o potencial da AutoDBLoader em contextos de data warehousing, integração de sistemas legados e rotinas de ETL.

Em síntese, a biblioteca atingiu plenamente seus objetivos, entregando uma solução eficiente e escalável. Para trabalhos futuros, destaca-se a implementação da atualização incremental de dados, que permitirá inserir apenas novos registros e atualizar os modificados. Assim, ampliando sua aplicabilidade em processos de sincronização entre bancos de produção e de réplica. Diante disso, a AutoDBLoader se

consolida como uma ferramenta promissora no ecossistema de engenharia e análise de dados, unindo automação, desempenho e confiabilidade.

7. Referências

DATE, C. J. A. **Guide to the SQL Standard**. 4. ed. [S.l.]: Addison-Wesley Longman Publishing Co., 1989.

DATE, C. J. **An Introduction to Database Systems**. 8. ed. Boston: Pearson/Addison Wesley, 2004.

ELMASRI, R.; NAVATHE, S. B. **Fundamentals of Database Systems**. 6. ed. Boston: Addison-Wesley, 2011.

ROMANO, F.; KRUGER, H. **Learn Python Programming: An In-Depth Introduction to the Fundamentals of Python**. 3. ed. Birmingham: Packt Publishing Ltd, 2021. p. 478-481.

RAMALHO, L. **Fluent Python**. 2. ed. Boston: O'Reilly Media, Inc., 2022.

REIS, J.; HOUSLEY, M. **Fundamentals of Data Engineering: Plan and Build Robust Data Systems**. 1. ed. Sebastopol: O'Reilly Media, 2022. p. 391-394.

PANDAS DEVELOPMENT TEAM. **Pandas Documentation**. [S.l.], 2025. Disponível em: https://pandas.pydata.org/docs/. Acesso em: 6 nov. 2025.

SQLALCHEMY TEAM. **SQLAlchemy Documentation**. [S.l.], 2025. Disponível em: https://docs.sqlalchemy.org/. Acesso em: 6 nov. 2025.

TIOBE.COM. **TIOBE Index**. [S.l.], 2023. Disponível em: https://www.tiobe.com/tiobe-index/. Acesso em: 12 nov. 2025.

KAGGLE. **Financial Transactions Dataset: Analytics**. [S.l.]: Kaggle, 2025. Disponível em: https://www.kaggle.com/datasets/computingvictor/transactions-fraud-datasets. Acesso em: 3 nov. 2025.

DB-Engines. **Popularity ranking of database management systems**. Disponível em: https://db-engines.com/en/ranking. Acesso em: 06 nov. 2025.

CARNEIRO, V. M. B. F.; MORAES, G.; BRAYNER, A. **I-DataMig: uma ferramenta inteligente para migração eficiente de bancos de dados**. In: **SIMPÓSIO BRASILEIRO DE BANCO DE DADOS (SBBD)**, 39., 2024, Florianópolis. *Anais...* Porto Alegre: Sociedade Brasileira de Computação, 2024. p. 89–94. DOI: 10.5753/sbbd_estendido.2024.240758. Acesso em: 6 nov. 2025.

NWOKEJI, J. C.; MATOVU, R. **A systematic literature review on big data extraction, transformation and loading (ETL)**. In: *Intelligent computing: proceedings*

of the 2021 Computing Conference. v. 2. Cham: Springer, 2021. DOI: 10.1007/978-3-030-80126-7_24. Acesso em: 6 nov. 2025.

HAMOUDA, S. **A conceptual model of semi-structured data for big data applications**. *Communications and Applied Information Systems Journal*, [S.l.], v. 5, n. 2, p. 20–28, 2024. Disponível em: https://www.iaras.org/home/caijels/a-conceptual-model-of-semi-structured-data-for-big-data-applications. Acesso em: 6 nov. 2025.

MACHADO, F. **Banco de Dados - Projeto e Implementação. Saraiva Educação S.A.**, 2018. ISBN 9788536509846. Disponível em: https://books.google.com.br/books?id=N4diDwAAQBAJ. Acesso em: 12 nov. 2025.