Fundação Getúlio Vargas
EMAp - Escola de Matemática Aplicada
Ciência de Dados e Inteligência Artificial
Computação Escalável

MICRO-FRAMEWORK: VENDA DE PASSAGENS AÉREAS

Guilherme Buss
Guilherme Carvalho
Gustavo Bianchi
João Gabriel
Vinícius Nascimento

Rio de Janeiro
2025

Sumário

1 Motivação

O objetivo deste projeto é desenvolver um micro framework em C++ voltado para a extração, processamento, transformação e carregamento (ETL) de dados de vendas de passagens aéreas. Como forma de otimização do nosso sistema, ele realiza benchmarks comparando o desempenho entre pipelines sequenciais e paralelos com diferentes quantidades de threads, mostrando a eficiência no uso destas.

2 Arquitetura Geral

2.1 Módulos

O sistema foi dividido em módulos/componentes:

Componente	**Responsabilidade Principal**
`DataFrame`	Representação tabular dos dados em memória
`Database`	Interface para criação e manipulação de tabelas no SQLite
`Extractor`	Extração de dados de arquivos JSON e CSV
`Handler`	Processamento e transformação de dados (validação, filtros, etc.)
`Loader`	Carga dos dados processados no banco SQLite
`Queue`	Comunicação thread-safe entre fases da pipeline paralela
`Series`	Representação unidimensional de dados com operações aritméticas e transformações
`ThreadPool`	Execução paralela com múltiplas threads
`Trigger`	Disparo de processamento com base em tempo ou requisição externa

2.2 Pipeline

O pipeline é composto por múltiplos componentes conectados em série e em paralelo, organizados em três grandes etapas: extração, transformação e carregamento.

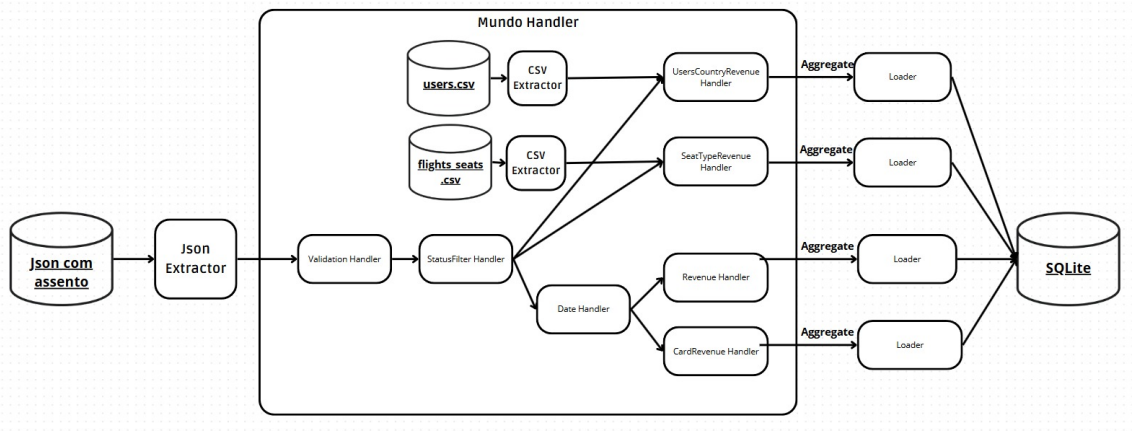

Figura 1: Arquitetura geral do pipeline de processamento e integração com SQLite

2.2.1 Extração

Os dados brutos são extraídos de diferentes fontes:

- **JSON com assento**: contém os dados das transações de venda de passagens. É processado por um componente `JsonExtractor`.
- **CSV** `users.csv`: contém dados de usuários, como país de origem.
- **CSV** `flights_seats.csv`: contém informações sobre assentos de voos e seus tipos.

2.2.2 Transformação

Após a extração, os dados passam por uma série de `Handlers` que compõem o núcleo do processamento lógico do sistema. Esses módulos operam de forma encadeada ou paralela, aplicando validações, filtros e agregações:

- **Validation Handler**: verifica se os dados do JSON são válidos para o processamento.
- **StatusFilter Handler**: filtra os dados com base no status da transação.
- **Date Handler**: normaliza ou filtra os dados por datas.
- **Revenue Handler**: calcula a receita total por venda.
- **CardRevenue Handler**: calcula a receita por tipo de cartão.
- **UsersCountryRevenue Handler**: correlaciona os dados de usuários com suas receitas por país, integrando o CSV `users.csv`.
- **SeatTypeRevenue Handler**: associa os tipos de assento à receita gerada por eles, usando o CSV `flights_seats.csv`.

2.3 Agregação e Carregamento

Após o processamento, os dados transformados são agregados em estruturas intermediárias e enviados para um componente `Loader`, que insere os dados consolidados no banco de dados `SQLite`. Essa arquitetura modular permite a execução paralela, favorecendo o desempenho e a escalabilidade do sistema, exatamente o que nos foi proposto para ser feito neste trabalho. O modelo é extensível, permitindo a inclusão de novos `Handlers`, transformações ou fontes de dados.

3 Estratégias de Processamento

3.1 Pipeline Sequencial

O pipeline sequencial é composto pelas seguintes etapas:

1. Validação
2. Filtro de status
3. Extração de data
4. Cálculo de receita
5. Agrupamento e carga no banco

3.2 Pipeline Paralelo

O pipeline paralelo divide os dados em `N` partes e aplica o mesmo fluxo em paralelo com uso de `ThreadPool`, utilizando `Queue` para comunicação entre etapas. Após o processamento, ocorre a agregação final para:

- Receita total
- Receita por cartão
- Receita por país do usuário

4 Triggers

O componente `Trigger` é responsável por iniciar a execução do pipeline de forma automatizada ou sob demanda. A interface é simples e genérica, suportando o registro de um *callback* (função de retorno) a ser chamado quando o disparo ocorrer. Duas implementações específicas foram desenvolvidas:

- `TimerTrigger`: dispara o callback periodicamente com base em um intervalo de tempo fixo, definido em milissegundos no construtor. Internamente, utiliza uma thread que executa um loop com `sleep_for()`, verificando a condição de execução e chamando o callback se apropriado. Essa abordagem é útil para pipelines agendados, como coletas periódicas de dados.
- `RequestTrigger`: permite o disparo manual via o método `trigger()`. É útil em cenários em que o pipeline deve ser executado sob demanda, como por meio de uma requisição ou evento do sistema.

A flexibilidade dessa estrutura permite que novos modos de disparo possam ser facilmente integrados ao sistema herdando da classe `Trigger`.

5 Benchmarking

Durante a execução, são registrados os tempos de:

- Processamento
- Carga no banco

As métricas são coletadas para os seguintes cenários:

Pipeline	**Threads**
Sequencial	1
Paralelo	4
Paralelo	8
Paralelo	12

Os tempos são armazenados na estrutura `TestResults` e impressos em formato tabular.

6 Resultados

Para avaliar o desempenho do pipeline desenvolvido, foi realizado um benchmark utilizando um arquivo JSON contendo 100 mil registros, chamado `ordersCemMil`. Os tempos foram medidos em milissegundos (ms) para as duas etapas principais: **Processamento** (handlers) e **Carga** (inserção em banco de dados SQLite).
Os testes consideraram três cenários:

Arquivo	Seq. Process	Seq. Load	Par. (4) Process	Par. (4) Load	Par. (8) Process	Par. (8) Load
ordersCemMil	66649	1264	6060	4547	1653	10212

Figura 2: Tempo de execução em milissegundos para o arquivo `ordersCemMil`.

Os resultados mostram uma melhoria significativa no tempo de processamento ao utilizar paralelismo:

- O tempo de **processamento** caiu de **66.649 ms** (sequencial) para **6.060 ms** com 4 threads e **1.653 ms** com 8 threads, representando uma redução de mais de 95%.
- O tempo de **carga no SQLite**, por outro lado, teve um comportamento diferente. Com 4 threads, houve ganho relevante (de **1.264 ms** para **4.547 ms**), mas com 8 threads, o tempo aumentou para **10.212 ms**. Isso se deve à natureza serializada do acesso ao banco SQLite, que não escala linearmente com mais threads concorrentes.