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.

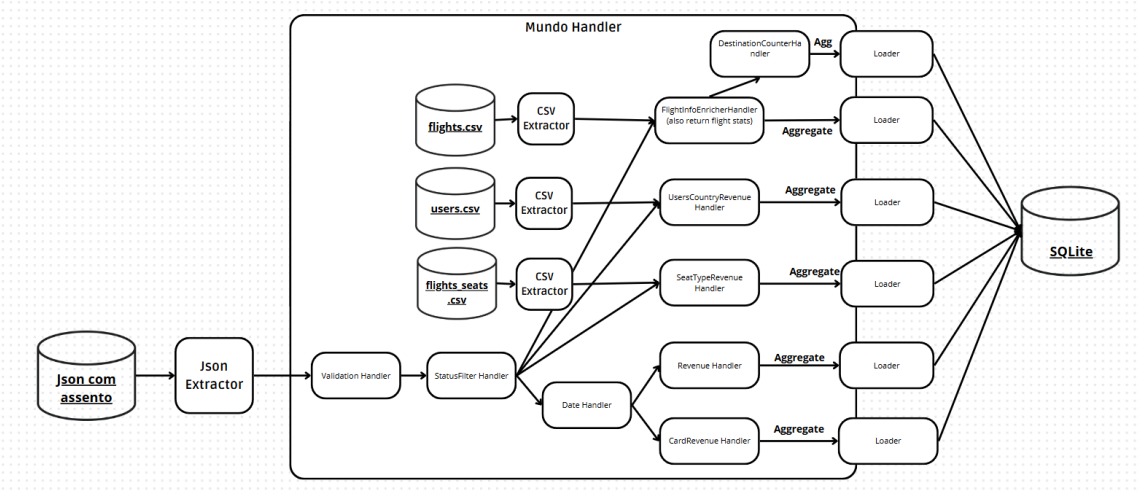


Figura 1: Arquitetura geral do pipeline de processamento e integração com SQLite

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 pelo componente `JsonExtractor`.

- **CSV** `flights.csv`: informações sobre os voos, extraídas por um `CSVExtractor`.
- **CSV** `users.csv`: contém dados de usuários, como país de origem.
- **CSV** `flights_seats.csv`: contém informações sobre os 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 e paralela, aplicando validações, filtros, enriquecimento e agregações:

- **Validation Handler**: verifica se os dados extraídos do JSON estão válidos para o processamento.
- **StatusFilter Handler**: filtra os dados com base no status da transação.
- **Date Handler**: normaliza ou filtra os dados por data da transação.
- **FlightInfoEnricher Handler**: enriquece os dados com informações dos voos, também calculando estatísticas relacionadas.
- **DestinationCounter Handler**: conta a quantidade de viagens por destino.
- **UsersCountryRevenue Handler**: correlaciona usuários com receitas por país, utilizando os dados do `users.csv`.
- **SeatTypeRevenue Handler**: associa os tipos de assento à receita gerada, utilizando `flights_seats.csv`.
- **Revenue Handler**: calcula a receita total por venda.
- **CardRevenue Handler**: calcula a receita total por tipo de cartão.

2.2.3 Agregação e Carregamento

Após o processamento e agregação dos dados, os resultados são encaminhados aos componentes `Loader`, responsáveis por inserir os dados transformados no banco de dados `SQLite`.
A arquitetura modular, com execução paralela dos fluxos, promove desempenho e escalabilidade. O modelo é extensível, permitindo a adição de novos `Handlers`, fontes de dados e lógicas de transformação conforme necessário.

3 Problemas e Soluções

Nesta seção descrevemos os principais desafios enfrentados durante o desenvolvimento da `main.cpp` e as soluções implementadas para garantir a robustez, performance e escalabilidade do microframework.

3.1 Carregamento Repetido de Dados Auxiliares

Problema: Cada execução de `processSequentialChunk` e `processParallelChunk` realizava a leitura de `users.csv`, `flights_seats.csv` e `flights.csv`, resultando em excesso de I/O e processamento desnecessário.
Solução:

- Carregar os *DataFrames* uma única vez no início do programa.
- Compartilhar os ponteiros usando `std::shared_ptr`, evitando cópias.
- Construir mapas auxiliares (`userIdToCountry`, `seatKeyToClass`) fora do laço de execução paralela.

3.2 Condições de Corrida na Impressão de Resultados

Problema: A saída das threads no console ficava embaralhada, dificultando a leitura da tabela.
Solução:

- Utilização de um `std::mutex table_mutex` global.
- Proteção das funções `printTableHeader()` e `printTableRow()` com `std::lock_guard<std::mutex>`.

3.3 Saturação do SQLite com Múltiplas Conexões

Problema: Gravações simultâneas no mesmo banco SQLite causavam contenção e lentidão.
Solução:

- Criar tabelas separadas para cada configuração de pipeline (ex.: `_4`, `_8`, `_12`).
- Considerar o uso de *batch inserts* ou migração futura para um SGBD com melhor suporte à concorrência (ex.: PostgreSQL).

3.4 Gargalo na Agregação Final

Problema: A agregação dos resultados parciais ocorria de forma sequencial, tornando-se um gargalo de desempenho.
Solução:

- Utilizar filas tipadas (`Queue<...>`) para segmentar os resultados.
- Concatenar os lotes antes de aplicar operações como `groupby()`, reduzindo o número de varreduras nos dados.

3.5 Falta de Flexibilidade na Extração de Esquemas

Problema: O extrator JSON possuía um esquema fixo, o que dificultava o reuso com diferentes estruturas de dados.
Solução:

- Permitir que o `Extractor` aceite um `std::vector<std::string>` com os nomes das colunas.
- Generalizar os métodos `extractChunk(...)` e `extractRandomChunk(...)` com parâmetros de esquema e tamanho.

3.6 Parada Lenta do `TimerTrigger`

Problema: Ao chamar `stop()`, o `TimerTrigger` aguardava o término do `sleep_for()`, atrasando a finalização.
Solução:

- Tornar a flag `running` uma variável atômica (`std::atomic<bool>`).
- Ao encerrar, definir `running = false` e chamar `timerThread.join()` para finalizar imediatamente após o intervalo atual.

4 Estratégias de Processamento

4.1 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

4.2 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

5 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`.

6 Benchmarking

Durante a execução, são registrados os tempos de:

- Processamento
- Carga no banco

As métricas são coletadas para os seguintes cenários:

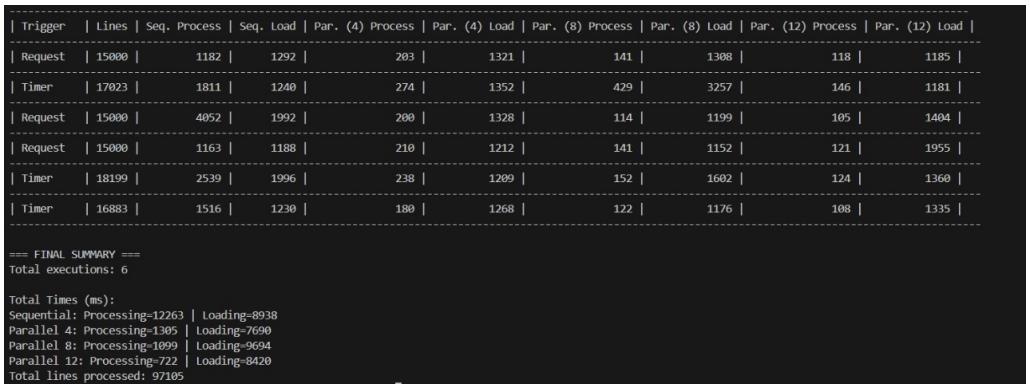
Pipeline	**Threads**
Sequencial	1
Paralelo	4
Paralelo	8
Paralelo	12

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

7 Resultados

7.1 Tempo

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:



Trigger	Lines	Seq. Process	Seq. Load	Par. (4) Process	Par. (4) Load	Par. (8) Process	Par. (8) Load	Par. (12) Process	Par. (12) Load
Request	15000	1182	1292	203	1321	141	1308	118	1185
Timer	17023	1811	1240	274	1352	429	3257	146	1181
Request	15000	4052	1992	200	1328	114	1199	105	1404
Request	15000	1163	1188	210	1212	141	1152	121	1955
Timer	18199	2539	1996	238	1209	152	1602	124	1360
Timer	16883	1516	1230	180	1268	122	1176	108	1335

```
=== FINAL SUMMARY ===
Total executions: 6

Total Times (ms):
Sequential: Processing=12263 | Loading=8938
Parallel 4: Processing=1305 | Loading=7690
Parallel 8: Processing=1099 | Loading=9694
Parallel 12: Processing=722 | Loading=8420
Total lines processed: 97105
```

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

Os resultados obtidos mostram claramente o impacto positivo do paralelismo na etapa de processamento, embora os ganhos na etapa de carregamento no banco (SQLite) sejam limitados:

- O tempo de **processamento** foi significativamente reduzido ao aplicar múltiplas threads: de **12.263 ms** na execução sequencial para **1.305 ms** com 4 threads, **1.099 ms** com 8 threads, e **722 ms** com 12 threads. Isso representa uma redução de até **94%**, evidenciando a eficiência da paralelização nessa etapa.
- Em contrapartida, o tempo de **carga no SQLite** apresentou um comportamento mais instável. Apesar de uma leve melhoria com 4 threads (**7.690 ms**), o tempo aumentou com 8 threads (**9.694 ms**) e oscilou com 12 threads (**8.420 ms**). Esse comportamento está relacionado às limitações de concorrência do SQLite, que tende a serializar operações de escrita, impactando negativamente a escalabilidade.

No total, foram processadas **97.015 linhas** em 6 execuções, demonstrando a robustez e consistência do pipeline proposto mesmo em cenários de carga elevada.

7.2 Estatísticas

Além dos benchmarks de desempenho, foi implementado um mecanismo de validação da consistência dos dados processados. Durante a geração sintética dos dados de entrada (especialmente no arquivo `ordersCemMil`), foram atribuídos **pesos probabilísticos** a diferentes categorias — como tipos de cartão, destinos e tipos de assento — influenciando diretamente a frequência com que cada categoria aparece nos dados. Após o carregamento no SQLite, é possível realizar consultas agregadas que permitem comparar as distribuições observadas com os pesos definidos no gerador, servindo como forma de verificação da integridade e coerência da pipeline de ETL. Esse processo garante que o sistema está respeitando corretamente as proporções dos dados simulados e preservando sua estrutura estatística ao longo de todas as etapas de extração, transformação e carga.

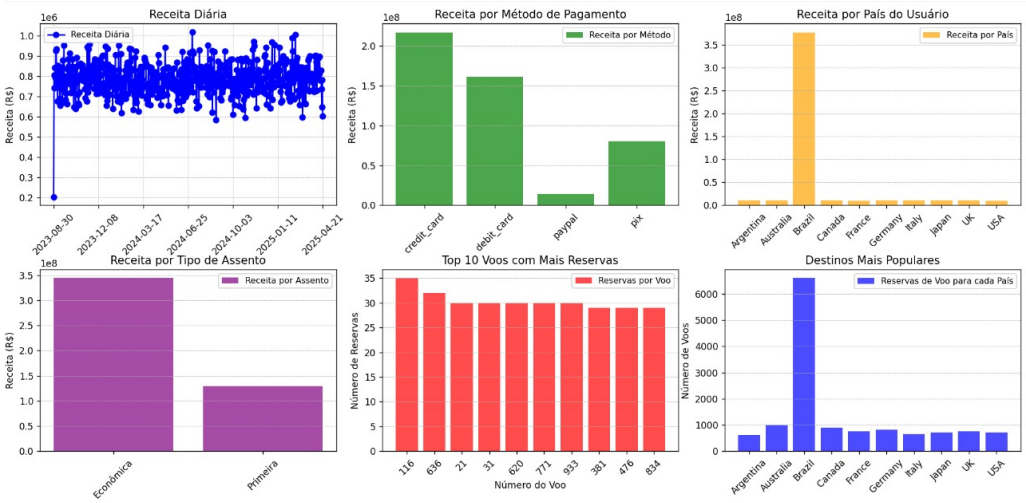


Figura 3: Gráficos resultantes com o tratamento dos handlers.

- A **receita diária** manteve-se relativamente estável ao longo do tempo, com variações naturais entre os dias.
- O **cartão de crédito** foi o método de pagamento mais utilizado, seguido por cartão de débito e Pix.
- O **Brasil** apresentou a maior receita entre todos os países, sendo também o principal destino em número de reservas.
- A **classe econômica** gerou significativamente mais receita do que a primeira classe.
- Os **10 voos mais reservados** tiveram entre 30 e 35 reservas, com boa distribuição de demanda entre eles.
- Entre os destinos, o **Brasil** liderou com folga em número de voos reservados, refletindo os pesos definidos no gerador de dados.

Esses resultados indicam que o pipeline de ETL está funcionando corretamente, preservando a estrutura estatística dos dados e refletindo fielmente os padrões esperados. Além disso, a visualização gráfica permite validar rapidamente a coerência e consistência das transformações realizadas durante o processo.