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. Para testar a otimização do sistema ele realiza testes comparando o desempenho entre pipelines sequenciais e paralelos com diferentes quantidades de threads, mostrando a eficiência no uso de cada uma para cada trigger ativado.

2 Arquitetura Geral

2.1 Módulos

O sistema foi dividido em módulos/componentes:

Componente	**Responsabilidade Principal**
`DataFrame`	Representação em tabela 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`	Carregar os dados processados no banco SQLite
`Queue`	Comunicação thread-safe entre as pipelines paralelas
`Series`	Representação unidimensional dos dados para uso na classe Datagrame
`ThreadPool`	Gestor das threads para execução paralela
`Trigger`	Disparo de processamento de pipelines com base em tempo ou requisição externa

2.2 Pipeline

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

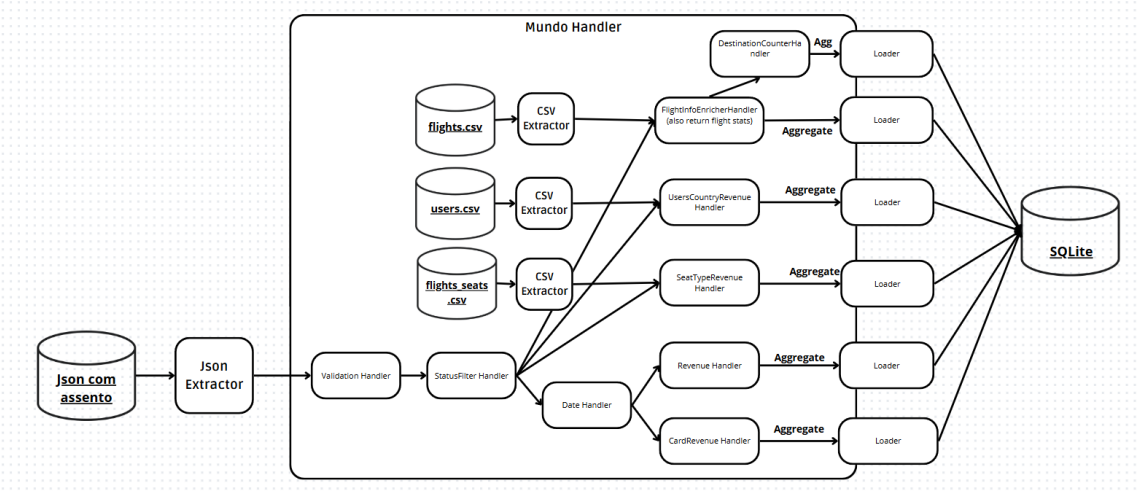


Figura 1: Arquitetura geral do pipeline de processamento e integração com SQLite

2.2.1 Extração

Os dados brutos são extraídos das seguintes 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 fazem os processamentos lógicos do sistema. Eles operam tanto de forma encadeada quanto paralela, aplicando nos dados 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 algumas 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 dos dados, os resultados são encaminhados ao `Loader`, responsável por inserir os dados transformados no banco de dados `SQLite`.

3 Problemas e Soluções

Nesta seção descrevemos os principais desafios que enfrentamos durante o desenvolvimento da `main.cpp` e as soluções implementadas para garantir a performance do micro-framework como desejávamos.

3.1 Salvamento no SQLite no paralelismo

Problema: Pra cada thread que terminava o pipeline ela salvava seu resultado no SQLite com o loader, deixando o tempo de load paralelo muito maior que o sequencial.
Solução:

- Não salvar no SQLite assim que termina a pipeline.
- Juntar todos os resultados e salvar uma única vez.

3.2 Uso do make_shared no acesso ao CSV

Problema: Quando acessava o CSV diretamente sem make_shared, quanto mais threads eram adicionadas, cada thread precisava fazer mais uma cópia do CSV, tornando o processamento mais lento ao invés de mais rápido.
Solução:

- Utilizar make_shared para criar um ponteiro compartilhado para o CSV
- Todas as threads passaram a acessar a mesma instância do CSV

3.3 Problemas no Mock do SQLite com Dados Fakes

Problema: Na implementação do mock do SQLite, ocorriam erros durante a inserção e leitura de dados fakes, principalmente porque o carregamento era linha a linha. Isso causava falhas durante as operações.
Solução:

- Implementação de um transaction do SQL para agrupar todas as operações de insert
- Substituição do processamento linha a linha por um commit único

3.4 Biblioteca do SQLite em C

Problema: A biblioteca que usamos do SQLite é em C e tinhamos problemas em alguns dos nosso computadores para rodar o código.
Solução:

- Introduzimos um MakeFile no código;
- Ele compila a biblioteca em um arquivo .o e roda o C++ de uma vez.

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 em paralelo realiza o mesmo processo porém ele divide os dados em `N` partes e aplica o mesmo fluxo para cada uma dessa partes em paralelo com uso de `ThreadPool`, utilizando `Queue` para comunicação entre as etapas. Após o processamento e agregação as tabelas resultantes serão:

- 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. O objetivo era simular um código que estivesse sendo usado de forma online. Implementamos dois tipos de trigger:

- `TimeTrigger`: dispara o callback periodicamente com base em um intervalo de tempo fixo porém com uma quantidade de dados aleatória para simular os dados que teriam chegado desde a última ativação de um trigger.
- `RequestTrigger`: Ocorre entre intervalos de tempo aleatório mas sempre com a mesma quantidade de dados, executaria no caso real quando essa quantidade de requisições tivesse acumulado na entrada.

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 de tabela.

7 Resultados

7.1 Tempo

Para avaliar o desempenho do pipeline desenvolvido, realizamos um benchmark utilizando um arquivo JSON contendo 190 mil registros, chamado `orders`. Os tempos foram medidos em milissegundos (ms) para as duas etapas principais - **Processamento** e **Carregamento** - e impressos assim que o pipeline associanado por aquele trigger terminasse:

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	1102	2974	127	2642	82	2460	67	2370
Timer	13616	862	3183	115	2844	65	2666	62	2562
Request	15000	805	1781	114	1766	74	2535	79	2105
Timer	36511	4844	2583	581	2054	242	1870	209	1836
Request	15000	1164	1778	136	1803	90	1877	83	1805
Timer	44755	7674	1634	641	1587	285	1644	240	1623
Timer	50118	7908	1796	932	1772	340	1807	243	1816

```
=== FINAL SUMMARY ===
Total executions: 7

Total Times (ms):
Sequential: Processing=24359 | Loading=15729
Parallel 4: Processing=2646 | Loading=14468
Parallel 8: Processing=1178 | Loading=14859
Parallel 12: Processing=983 | Loading=14117
Total lines processed: 190000
```

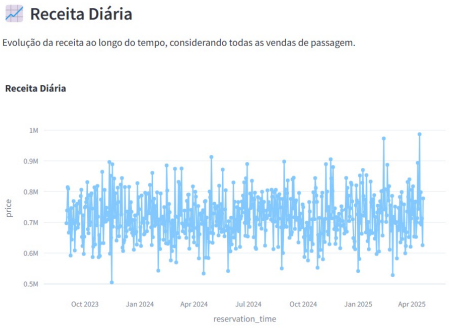
Figura 2: Tempo de execução em milissegundos para o arquivo `orders`.

Os resultados obtidos mostram claramente o impacto positivo do paralelismo no processamento, o tempo de **processamento** foi significativamente reduzido ao aplicar múltiplas threads: de **24.359 ms** na execução sequencial para **2.646 ms** com 4 threads, **1.178 ms** com 8 threads, e **983 ms** com 12 threads. Isso representa uma redução de até **96%**, evidenciando a eficiência da paralelização nessa etapa.

7.2 Estatísticas

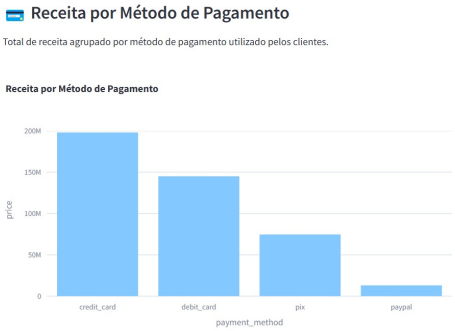
Para visualização utilizamos o python para fazer o Dashboard e tentamos escolher algumas visualizações que poderiam agregar valor num contexto real de uma empresa de passagem aérea.
Para que as visualizações fizessem algum sentido na geração dos dados o grupo escolheu algumas categorias para ter maior quantidade de dados, influenciando também as probabilidade na hora da geração, assim, no nosso dashboard é possível verificar se após as operações os dados se mantiveram consistentess ou não, servindo como forma de verificação da integridade e coerência da pipeline.
Nossos resultados obtidos foram:

- Receita diária;



A **receita diária** manteve-se relativamente estável ao longo do tempo, com variações naturais entre os dias.

- Receita por método de pagamento;



O **cartão de crédito** foi o método de pagamento mais utilizado, seguido por cartão de débito e Pix.

- Receita por país do usuário;

O **Brasil** apresentou a maior receita entre todos os países, sendo também o principal destino em número de reservas.

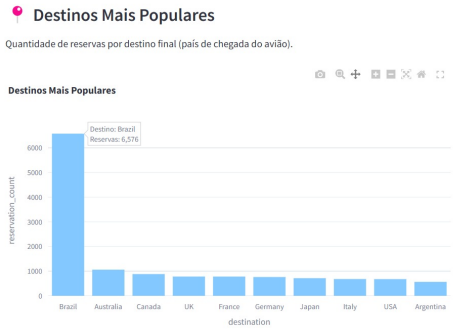
- Receita por tipo de assento;

A **classe econômica** gerou significativamente mais receita do que a primeira classe.

- Top 10 Voos com mais reservas;

Os **10 voos mais reservados** tiveram entre 30 e 35 reservas, com boa distribuição de demanda entre eles.

- Destinos mais populares.



Entre os destinos, o **Brasil** liderou com folga em número de voos reservados, refletindo os pesos definidos no gerador de dados.

Todas situações que fariam sentido num contexto real - que mantém consistência também com as probabilidades que escolhemos ao gerar os dados - e informações que uma empresa possivelmente iria querer saber sobre seu fluxo de negócios.