Relatório trabalho Computação Escalável:

Plataforma de E-commerce e Gestão de Cadeia de Suprimentos

21 de Abril de 2025

Integrantes:

Daniel de Miranda Almeida

João Felipe Vilas Boas

Lívia Verly

Luís Felipe de Abreu Marciano

Paulo César Gomes Rodrigues

Visão Geral

Este trabalho propõe o desenvolvimento de um micro-framework para a construção de pipelines de processamento de dados, com foco em eficiência e suporte a paralelismo. A proposta é criar uma solução escalável, considerando a capacidade de processamento da máquina do usuário, e que permita o processamento de múltiplos componentes de forma concorrente. Para demonstrar sua aplicabilidade, será utilizado um projeto de exemplo nas áreas de E-commerce e Gestão da Cadeia de Suprimentos. É possível ver todas implementações que serão explicadas neste relatório em nosso repositório GitHub (https://github.com/Vilasz/A1_comp_esc).

Objetivos

O principal objetivo deste trabalho é demonstrar, de forma eficiente, como o processamento de dados pode ser potencializado por meio do uso de técnicas de paralelismo e concorrência em uma única máquina. A proposta é evidenciar os ganhos de desempenho obtidos a partir da execução simultânea de tarefas em diferentes núcleos de processamento. Para isso, será desenvolvido um micro-framework que incorpora esses conceitos, permitindo a construção de pipelines modulares e escaláveis.

Python ou C++?

No começo do projeto, tivemos que decidir qual linguagem de programação usar. Escolhemos o Python por questão de afinidade do grupo com a linguagem, e também por conta das aulas sobre técnicas de paralelismo e concorrência usando Python.

É verdade que o C++ pode ser mais rápido e permite um controle maior sobre o uso da memória e a sincronização entre tarefas. No entanto, para os objetivos do nosso trabalho, o Python foi a melhor opção por conta da maior produtividade e atende bem às nossas necessidades, visto que não queremos que o programa seja ótimo, mas que se torne mais eficiente com a utilização de paralelismo e concorrência. Além disso, ele conta com bibliotecas que contribuem

para o trabalho com paralelismo, como multiprocessing, concurrent.futures e threading.

Mas o Python tem uma limitação importante quando o programa exige muito processamento (CPU-bound): o Global Interpreter Lock (GIL). O GIL impede que várias threads executem código Python ao mesmo tempo dentro do mesmo processo, o que limita o uso eficiente de processadores com vários núcleos. Para contornar esse problema, usamos multiprocessing (processos em vez de threads) e pools de processos, que permitem uma execução paralela real, superando a barreira do GIL.

Além disso, implementamos um modelo híbrido usando o HybridPool, que alterna entre ProcessPoolExecutor (para tarefas pesadas em CPU) e ThreadPoolExecutor (para tarefas que envolvem espera, como operações de I/O). Essa estratégia ajudou a equilibrar melhor os recursos, aproveitando ao máximo o hardware durante os testes e garantindo que nosso sistema fosse escalável e eficiente.

Micro-framework

Nosso micro-framework foi criado como uma base para facilitar a construção de pipelines paralelos e concorrentes para processamento de dados. Ele inclui suporte para estruturas como DataFrames, filas com controle de concorrência, sistemas de tarefas com dependências (DAGs) e ferramentas para agendamento e monitoramento.

Principais componentes:

- **DataFrame e Series:** São estruturas de dados baseadas nas estruturas do pandas. Elas permitem operações como soma, média, filtragem, ordenação e agrupamento (groupby), sendo úteis para manipulação eficiente de dados.
- **Queue e MapMutex:** São filas inteligentes com controle de capacidade e mecanismos de sincronização (como Condition), evitando problemas como condições de corrida quando vários processadores acessam os dados ao mesmo tempo. O MapMutex permite bloquear operações por chave, garantindo que apenas um worker por vez modifique um dado específico.
- **ThreadWrapper e Handler:** Classes básicas para criar workers (trabalhadores concorrentes) e roteadores que direcionam dados entre filas

de entrada e saída. Elas contribuem para organizar o fluxo de processamento de forma modular.

- **Trigger, Scheduler e HybridPool:** Ferramentas para acionar e gerenciar tarefas automaticamente. O HybridPool é responsável por distribuir tarefas entre threads e processos de forma inteligente, de acordo com o limite máximo de uso da CPU (CPU_LIMIT), o que melhora o desempenho em tarefas mistas (CPU-bound e I/O-bound).

O micro-framework serve como base para um pipeline robusto e escalável. Ou seja, você consegue montar um processamento de dados eficiente com pouco código e alta flexibilidade.

Paralelismo e Concorrência no Framework

Nosso framework foi idealizado para se beneficiar da agilidade da concorrência com threads e do paralelismo real com processos. Essa abordagem híbrida permite que o sistema se adapte automaticamente ao tipo de cada tarefa, garantindo máxima eficiência em todo o pipeline de processamento.

Quando o trabalho envolve operações de entrada e saída, como ler arquivos, fazer consultas a bancos de dados SQLite ou acessar serviços externos, o framework prioriza o uso de threads. Isso porque essas operações passam a maior parte do tempo aguardando respostas, e as threads conseguem lidar muito bem com esse tipo de tarefa sem serem limitadas pelo GIL do Python.

Por outro lado, quando em tarefas que exigem intenso poder de processamento, como cálculos de métricas com muitos dados, o framework delega o trabalho para processos independentes. Eles rodam livres do GIL, podendo aproveitar todos os núcleos de processamento disponíveis na máquina. Isso significa que uma operação complexa pode ser dividida em várias partes e processada em paralelo, reduzindo significativamente o tempo total de execução.

O responsável pela distribuição de tarefas é o HybridPool. Ele é um gerenciador que monitora constantemente o fluxo de trabalho. Ele analisa o tipo de cada tarefa, a carga atual do sistema e a disponibilidade de recursos para decidir dinamicamente se usa um ProcessPoolExecutor (para paralelismo verdadeiro) ou um ThreadPoolExecutor (para concorrência eficiente). Essa tomada de decisão automática evita que o sistema seja sobrecarregado com processos desnecessários.

Em uma abordagem anterior na construção do framework utilizamos o multiprocessing.Process. É possível construir um pipeline inteiro usando diretamente multiprocessing.Process, mas exigiria muito mais código e complexidade.. Além disso, a cada tarefa seria necessário criar e destruir um novo processo, o que é extremamente custoso em termos de tempo e uso de memória.

Por isso, utilizamos o ProcessPoolExecutor, que encapsula toda essa complexidade. Ele reutiliza processos existentes, gerencia filas e sincronização automaticamente, e torna o paralelismo mais eficiente e produtivo.

Além dessa capacidade central de distribuição inteligente, o framework incorpora um avançado sistema de agendamento baseado em grafos de dependência (DAG). Isso permite definir relações complexas entre tarefas, garantindo que cada etapa só seja executada quando todas suas dependências predecessoras tiverem sido concluídas com sucesso. Imagine um pipeline onde a etapa de limpeza de dados precisa esperar a conclusão da extração, e a análise estatística só pode começar após a limpeza estar pronta - tudo isso é gerenciado automaticamente pelo framework.

Outro fator é o sistema de coleta de métricas. Cada etapa do pipeline gera dados detalhados sobre seu desempenho, incluindo tempos de execução, taxas de processamento e utilização de recursos. Essas métricas são coletadas em tempo real e podem ser visualizadas em dashboards.

Resumo do passo a passo do pipeline

O funcionamento do pipeline segue uma lógica clara e eficiente:

1. O processo principal carrega e divide os dados em chunks (blocos).
2. Para cada chunk, o pipeline chama HybridPool.submit():
    - Se o número de processos ativos ainda for menor que o limite definido, a tarefa é enviada para o ProcessPoolExecutor, que cria (via fork ou spawn dependendo do OS) um novo processo e executa a função _process_chunk.
    - Se o número de processos já atingiu o limite, a tarefa é enviada para o ThreadPoolExecutor, que usa threads para lidar com tarefas restantes sem abrir novos processos.
3. A função _process_chunk realiza um processamento em paralelo de fato, pois roda em processos separados.

4. Por fim, a função as_completed() recolhe os resultados à medida que cada tarefa termina, permitindo a consolidação dos dados de forma eficiente e ordenada.

Projeto Exemplo: Plataforma de E-commerce e Gestão de Cadeia de Suprimentos

Fizemos um pipeline de processamento de dados para um sistema de e-commerce com gestão logística. Esse cenário foi escolhido por representar desafios reais que podem envolver grande volume de informações, diversos tipos de dados e a necessidade de processamento rápido.

O sistema começa com um banco de dados SQLite alimentado por dados fictícios. Usamos a biblioteca Faker para criar clientes, produtos, pedidos e informações de logística.

O processo principal está no arquivo mvp_pipeline.py, que executa todas as etapas de forma organizada:

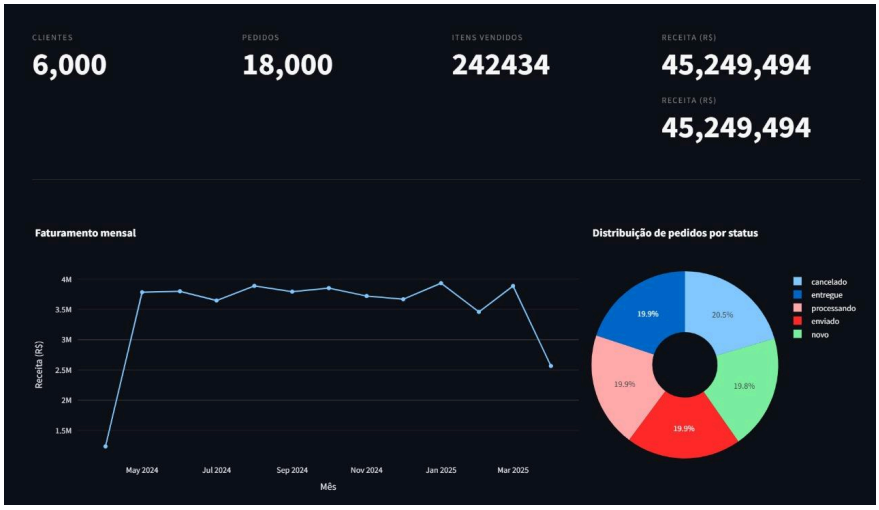
Primeiro, extraímos os dados brutos em formatos como CSV e JSON. Em seguida, vem a fase de transformação: dividimos os dados em blocos e processamos em paralelo, calculando métricas como vendas por produto, receita por canal de venda e volume por estado. Incluímos até mesmo alguns cálculos propositalmente pesados para testar os limites do sistema.

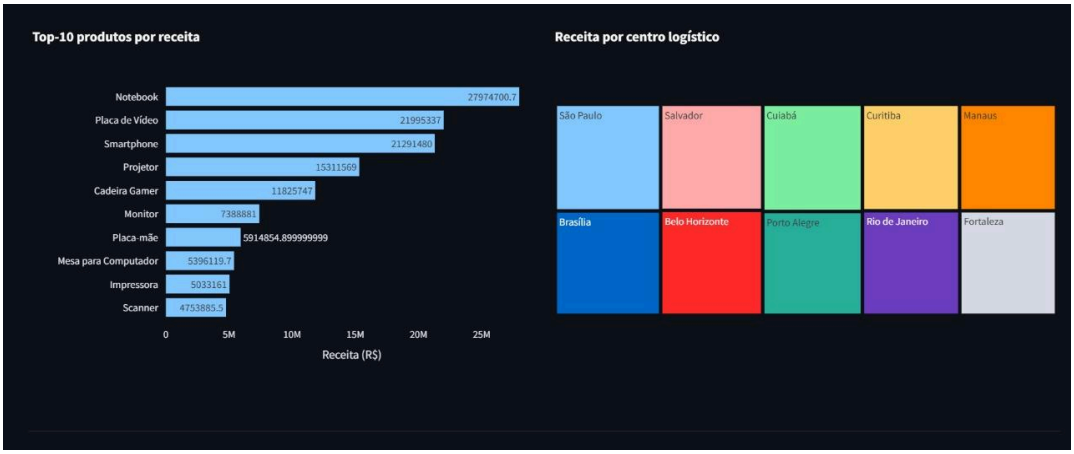
Os resultados são armazenados em tabelas resumidas no banco de dados. Por fim, geramos um relatório completo com as principais métricas de desempenho e análises comerciais.

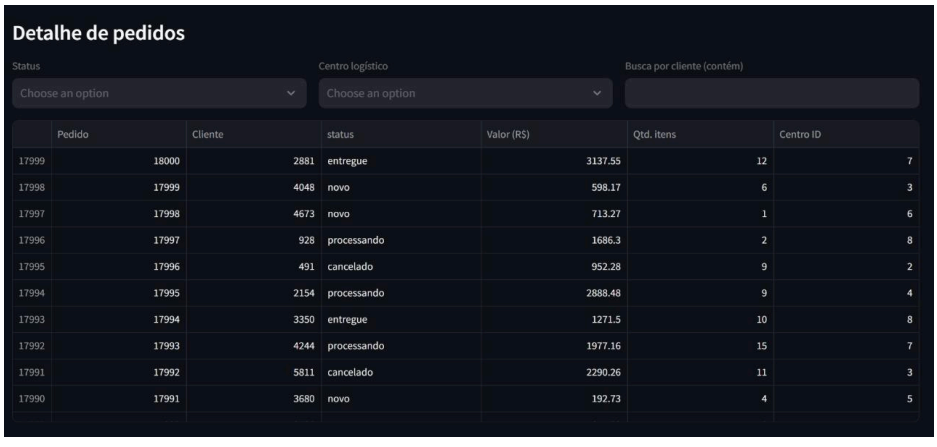
Vale ressaltar que o objetivo do nosso trabalho é demonstrar como o processamento de dados pode se tornar muito mais eficiente com a aplicação de técnicas de paralelismo e concorrência. Por isso, não nos preocupamos em tornar cada função dos tratadores a mais otimizada possível. Na verdade, podemos considerar até desejável que essas funções realizem um processamento mais intenso, pois isso evidencia melhor o ganho de desempenho proporcionado por uma arquitetura paralela, à medida que aumentamos o número de processos executados simultaneamente.

Resultados

O dashboard de saída do nosso pipeline de exemplo ficou da seguinte forma:







Detalhe de pedidos

Status: Choose an option

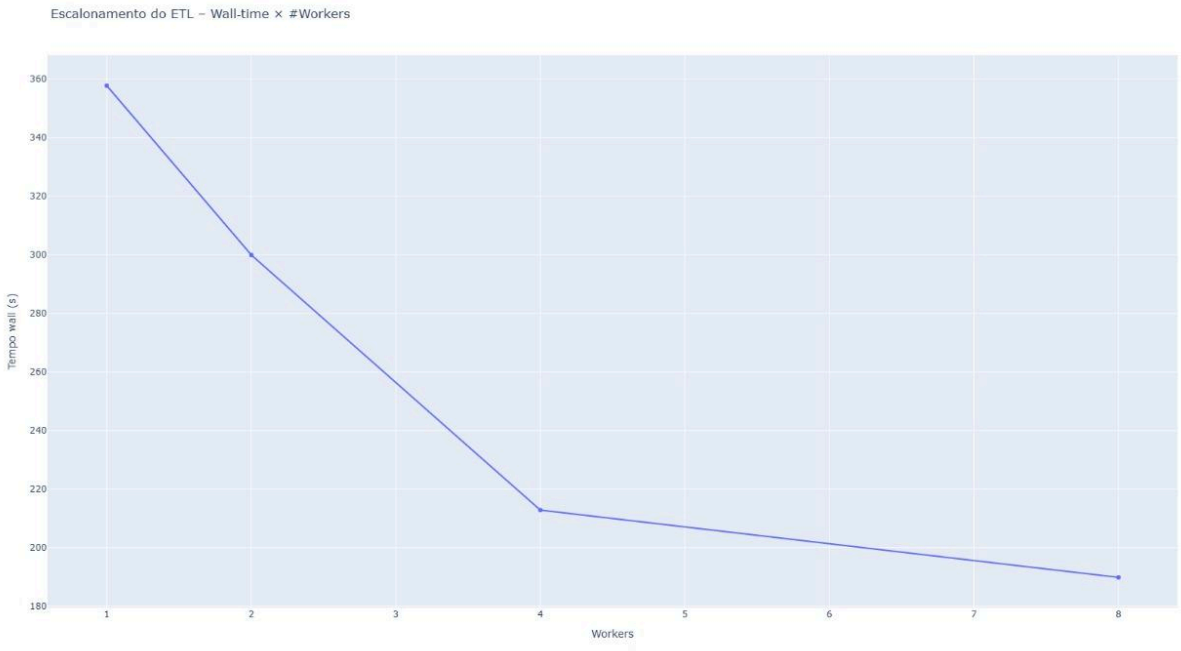
Centro logístico: Choose an option

Busca por cliente (contém)

	Pedido	Cliente	status	Valor (R$)	Qtd. itens	Centro ID
17999	18000	2881	entregue	3137.55	12	7
17998	17999	4048	novo	598.17	6	3
17997	17998	4673	novo	713.27	1	6
17996	17997	928	processando	1686.3	2	8
17995	17996	491	cancelado	952.28	9	2
17994	17995	2154	processando	2888.48	9	4
17993	17994	3350	entregue	1271.5	10	8
17992	17993	4244	processando	1977.16	15	7
17991	17992	5811	cancelado	2290.26	11	3
17990	17991	3680	novo	192.73	4	5

Na segunda imagem há um hover nos quadrados referentes aos estados que mostra a receita em cada um deles.

Agora vamos fazer uma análise do impacto que tem o número de workers (processos) no tempo de execução do pipeline.

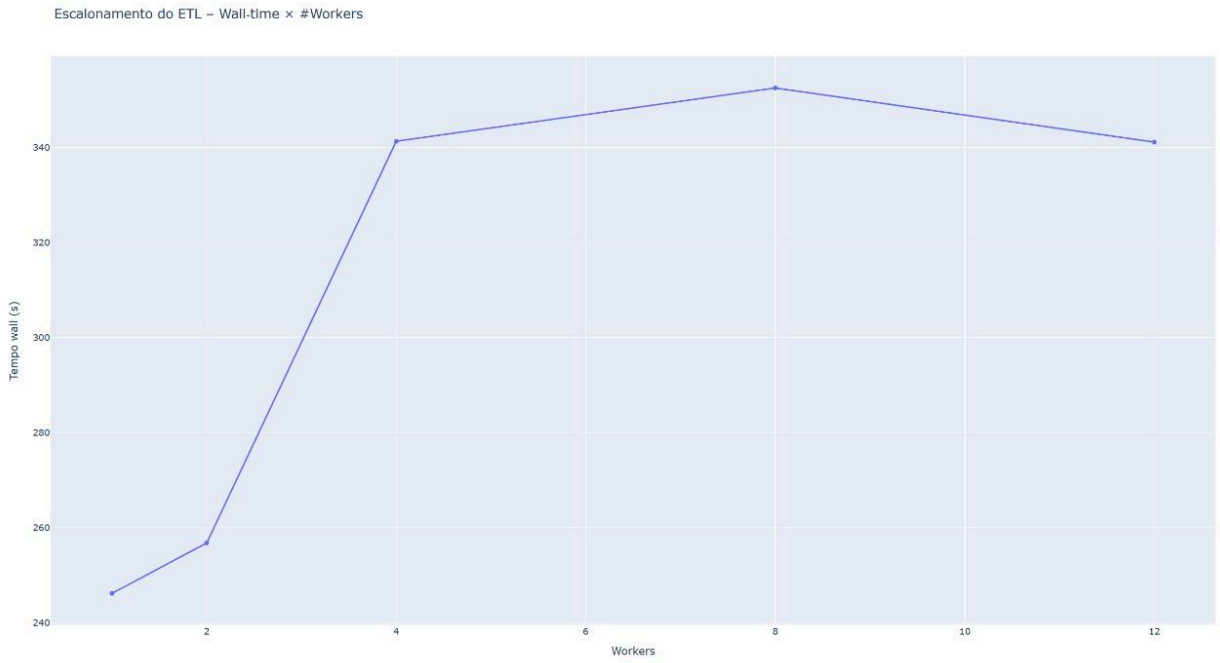


No gráfico acima, podemos ver o tempo total de execução do pipeline (tempo wall) em função da quantidade de workers utilizados. Os testes foram feitos com grande quantidade de dados, 700 mil linhas, juntando o banco de dados com o mock. O resultado mostra que, conforme aumentamos o número de workers, o tempo de execução diminui bastante no começo, principalmente entre 1 e 4 workers. Depois disso, o ganho de desempenho continua, mas de forma mais lenta.

Isso mostra que o nosso pipeline está conseguindo aproveitar bem o paralelismo, especialmente quando o volume de dados é maior. No entanto, também fica claro que há um ponto em que adicionar mais workers não melhora tanto, o que é esperado, já que existe um limite na quantidade de núcleos disponíveis e também um custo de coordenação entre as tarefas.

Esses testes comprovam que o uso de múltiplos processos realmente acelera o processamento, validando a proposta do nosso micro-framework de ser eficiente em cenários com grandes volumes de dados.

Outra análise interessante a ser feita é observar como é o desempenho do pipeline com uma quantidade menor de dados, 12 mil linhas, por exemplo, à medida que o número de workers aumenta.



É possível perceber que ele teve um desempenho não muito bom quando usamos poucos dados. Isso acontece porque o pipeline tem uma estrutura mais pesada, com várias etapas de preparação, como criação de filas, divisão dos dados em blocos e controle de processos.

Quando os dados são poucos, o tempo gasto nessas preparações acaba sendo maior que o tempo necessário para processar os dados em si. Por isso, o desempenho parece pior. E isso é cada vez mais acentuado à medida que aumentamos o número de workers, visto que isso implica um aumento do custo fixo para criar e lidar com múltiplos processos. Já quando usamos muitos dados, esse tempo de preparação se dilui, e o pipeline consegue aproveitar melhor os vários núcleos do processador, rodando várias partes ao mesmo tempo.

Isso mostra que o nosso sistema funciona melhor em cenários com bastante informação para processar, o que é comum em aplicações reais. Nessas situações, conseguimos ver ganhos claros de desempenho com o uso de paralelismo.