Computação Escalável

Análise de Sistema de Reservas de Viagens

Gustavo Tironi, Kauan Mariani Ferreira, Matheus Fillype Ferreira de Carvalho,
Pedro Henrique Coterli, Sillas Rocha da Costa

22 de abril de 2025

Conteúdo

1 Introdução **3**

2 Modelagem Geral **3**

3 Decisões de Projeto **4**

3.1 Arquitetura de Execução 5

3.2 Comunicação com Buffers e Execução 6

3.3 Resumo do Fluxo de Execução 6

3.4 Outras decisões importantes 7

3.5 Triggers 7

3.6 Criação de Estatísticas com o Pipeline 8

4 Problemas e Soluções **9**

4.1 Controle de Concorrência com Buffer 9

4.1.1 Sinalização de Término 9

4.2 Suporte a Múltiplos Buffers de Entrada 10

4.2.1 Solução Implementada 10

4.2.2 Grau de Complexidade da Solução 10

4.3 Problema ao Realizar `group_by` em um Transformer 11

4.3.1 Solução Implementada 11

5 Vantagens e desvantagens das escolhas 12

5.1 Vantagens 12

5.2 Desvantagens 13

5.3 Trade-off 14

6 Projeto Exemplo 14

7 Resultados Experimentais 15

8 Conclusão 18

1 Introdução

Neste trabalho, propomos um framework modular e extensível voltado à construção de pipelines de dados com alto grau de paralelismo. Nosso objetivo é abstrair a complexidade do gerenciamento de concorrência, permitindo que o usuário concentre seus esforços na lógica de transformação dos dados, enquanto o sistema se encarrega de distribuir a carga de trabalho de forma eficiente.

A arquitetura proposta foi projetada para ser flexível, permitindo o uso de diferentes fontes de dados, múltiplos estágios de processamento e personalização da lógica em pontos estratégicos do pipeline. Utilizamos uma combinação de fila de tarefas, pool de threads e buffers de entrada e saída para garantir execução paralela segura e eficiente.

O tema escolhido pelo nosso grupo foi o sistema de reservas de viagens. Conforme a orientação do professor, atuamos como analistas de dados, voltando nosso foco para a criação de um fluxo que receba os dados fornecidos pela empresa (simulados), processe-os por meio da pipeline desenvolvida e, a partir disso, gere análises relevantes para o negócio. A proposta busca alinhar eficiência técnica com geração de valor prático, simulando um cenário real de análise de dados em larga escala.

Ao longo deste documento, detalhamos as decisões de projeto, a arquitetura de execução, os benefícios e limitações do modelo adotado, além de apresentar um projeto exemplo e os resultados experimentais que comprovam a eficácia da solução.

2 Modelagem Geral

Todo o microframework foi estruturado sobre a implementação de uma abstração central: o **DataFrame**.

Foram implementados extratores para diferentes fontes de dados, incluindo arquivos `.txt`, `.csv` e `.db`. Esses extratores transformam os arquivos em objetos do tipo DataFrame, permitindo o uso padronizado e modular dos dados ao longo da aplicação. Esse extrator também tem sua execução como parte do framework e, portanto, conta com o controle de concorrência e balanceamento, sendo executado em conjunto com os outros componentes.

O fluxo de processamento é baseado em três conceitos principais:

- **Extratores**: responsáveis por carregar ou gerar os dados que serão processados.
- **Tratadores**: realizam o processamento efetivo dos dados, podendo aplicar filtros, agrupamentos, transformações, etc.

- **Loaders**: responsáveis por armazenar ou exportar os dados resultantes, seja em arquivos ou em bancos de dados.

O **DataFrame** oferece uma série de métodos nativos que podem ser utilizados ou customizados nos tratadores, como operações de `groupby`, `filter`, agregações, entre outros. Isso proporciona uma API unificada para manipulação de dados que pode ser facilmente reutilizada em diferentes contextos analíticos, como pedido no projeto.

O centro do sistema é um componente chamado **manager**, responsável por orquestrar toda a execução de forma balanceada, paralela e com controle de concorrência. Ele gerencia a comunicação entre os componentes, a distribuição das tarefas entre threads, o controle de concorrência e a sincronização dos resultados. O usuário, ao utilizar o framework, apenas precisa se preocupar com os componentes e a quantidade de threads desejada, sendo todo o restante controlado pelo manager de forma automática.

3 Decisões de Projeto

A principal decisão de projeto adotada foi a forma de lidar com o balanceamento de carga e a concorrência entre os módulos do framework. No contexto proposto, nos deparamos com uma variação clássica do problema do produtor-consumidor (ou leitor-escritor), o que exigiu soluções que garantissem paralelismo eficiente.

Para isso, seguimos os seguintes princípios orientadores:

- O sistema deve ser capaz de processar múltiplos leitores e escritores em paralelo, evitando gargalos. Todas as etapas do pipeline devem ser capazes de ser executadas simultaneamente.
- A estrutura precisa suportar pipelines complexos e com múltiplos estágios, inclusive na presença de zonas de conflito de acesso aos dados.
- O gerenciamento de concorrência e balanceamento de carga deve ser totalmente abstraído do usuário. O usuário apenas define os componentes e a quantidade de threads disponíveis para execução.
- Todo o poder computacional disponibilizado pelo usuário deve ser utilizado da forma mais eficiente possível durante toda a execução.

3.1 Arquitetura de Execução

Para atender a esses requisitos, o sistema foi implementado com base em uma arquitetura composta por uma **fila de tarefas** (*task queue*) e uma **pool de threads** (*thread pool*). A execução segue o seguinte modelo: cada componente do pipeline é responsável por criar descrições de tarefas (tasks), em vez de executá-las diretamente.

Cada tarefa contém:

- A função que deve ser executada;
- O endereço dos dados de entrada;
- O endereço onde os resultados devem ser armazenados.

Essas tarefas são então inseridas em uma fila compartilhada de execução, conforme ilustrado na Figura 1.

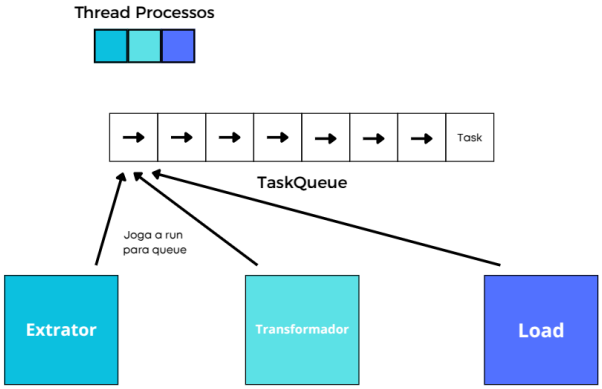

Figura 1: Fila de Tarefas

Cada componente possui uma **thread auxiliar dedicada** exclusivamente à criação de tarefas. Essas threads auxiliares são distintas da thread pool principal utilizada para execução. Por exemplo, se o pipeline possui 10 componentes e o usuário define o uso de 4 threads, o sistema criará 14 threads **além da principal**: 10 para agendamento e 4 para execução (totalizando 15 com a main).

Essas threads auxiliares não executam computações pesadas, elas apenas realizam a captura dos dados de entrada, preparam a tarefa e a colocam na fila. O custo computacional dessa etapa é, portanto, muito baixo.

3.2 Comunicação com Buffers e Execução

Para possibilitar a execução paralela entre os componentes, fugindo da lógica linear, cada componente pode possuir:

- **buffers de entrada**, de onde os dados são lidos;
- **buffers de saída**, para onde os resultados são escritos.

No nosso caso, os componentes extratores possuem apenas buffers de saída (a entrada são as próprias fontes de dados), os transformadores possuem ambos os tipos e os carregadores (loaders) apresentam apenas buffers de entrada (apenas 1 inclusive, pois seu papel é apenas receber esse DataFrame e salvá-lo externamente ou exibi-lo).

Esses buffers são criados e controlados pelo próprio framework, de acordo com as especificações do usuário.

A execução real das tarefas é realizada por uma **thread pool**, responsável por consumir as tarefas da fila e executar as funções especificadas. Ela será a responsável por usar os dados dos buffers de entrada e carregar os resultados nos buffers de saída. A Figura 2 ilustra como a thread pool interage com os buffers e controla a execução.

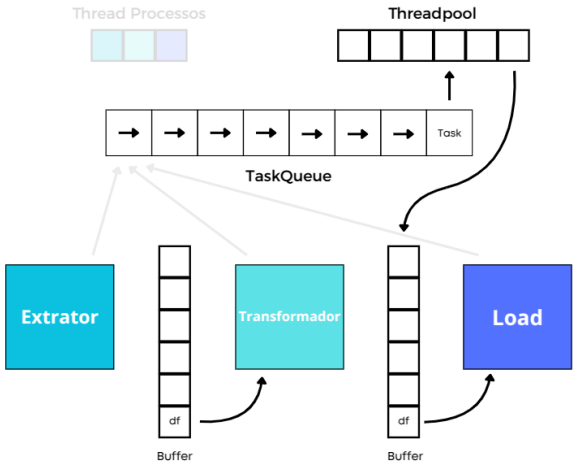

Figura 2: Execução de tarefas pela Thread Pool

3.3 Resumo do Fluxo de Execução

Resumidamente, o funcionamento do sistema segue os seguintes passos:

1. Cada componente possui uma thread auxiliar que cria uma tarefa descrevendo a função a ser executada, os dados de entrada e o destino dos resultados.

2. A tarefa é colocada na fila de tarefas.

3. A thread pool consome as tarefas da fila e realiza a execução, interagindo diretamente com os buffers indicados, jogando num buffer de output ou printando no caso de ser um loader.

Dessa forma, o framework consegue manter um alto grau de paralelismo e eficiência.

3.4 Outras decisões importantes

Foi criado um extrator genérico utilizando o padrão de projeto ***Strategy***, permitindo que o framework funcione com diferentes fontes de dados, como arquivos `.csv` e bancos de dados `.db`.

A customização do tratamento dos dados pelo usuário (*hot-spot*) é feita por meio da modificação do método `run()` em uma classe tratador herdada. Isso permite aplicar transformações específicas ao `Dataframe`, conforme necessário.

A seguir (Listing 1), apresentamos um exemplo de uso dessa abordagem:

```
class filter_hotel : public Transformer<Dataframe> {
public:
    using Transformer::Transformer; // Herda o construtor

    Dataframe run(std::vector<Dataframe*> input) override {
        // Definicao do tratador pelo usuario
        Dataframe df_filtered = (*input[0]).filtroByValue("ocupado", 0);
        return df_filtered;
    }
};
```

Listing 1: Exemplo de uso do tratador

3.5 Triggers

Para garantir maior autonomia e responsividade à execução da pipeline, foram desenvolvidos dois mecanismos de ativação distintos: o ***TimeTrigger*** e o ***EventTrigger***.

O **TimeTrigger** é responsável por acionar a pipeline de forma periódica, com base em um intervalo de tempo definido pelo usuário. Ele executa uma função passada como parâmetro a cada ciclo de tempo, descontando automaticamente o tempo gasto na execução da

função para manter a regularidade do disparo. Sua implementação utiliza `std::thread`, `std::chrono` e controle por `std::atomic` para garantir segurança em execução concorrente em meio a toda a estrutura desenvolvida ao longo do trabalho.

Já o **EventTrigger** foi projetado para monitorar alterações em um arquivo específico — por exemplo, um arquivo de base de dados ou log. A cada intervalo de tempo (polling), ele verifica se houve mudança no tempo de modificação do arquivo. Caso detecte uma alteração, a função fornecida é imediatamente executada. O monitoramento é feito com base na biblioteca `<filesystem>` e também emprega multithreading para operar em segundo plano.

Ambos os triggers podem ser iniciados e interrompidos de forma controlada por métodos públicos, proporcionando flexibilidade e integração direta com outros componentes do sistema. A separação entre os dois tipos permite que o sistema reaja tanto a eventos externos quanto funcione de forma programada, cobrindo assim diferentes cenários operacionais.

3.6 Criação de Estatísticas com o Pipeline

A arquitetura do nosso *pipeline* permite que o usuário desenvolva estatísticas personalizadas de maneira simples. As classes `Transformer<Dataframe>`, permitem a criação de estatisticas por meio do override da função `calculateStats` e assim o usuário consegue criar qualquer estatistica que seja um inteiro ou floar e então pegar essa estatistica no final.

A função `calculateStats` mostra como é possível aplicar filtros e calcular métricas com poucas linhas de código. A seguir, apresentamos um exemplo da saída produzida por esse módulo:

```
Número de quartos ocupados em toda a base: 104807
Número de quartos não ocupados em toda a base: 157199
Número de quartos no Rio de Janeiro: 14526
Número de quartos em Campo Grande: 5380
Número de cidades destino diferentes em toda a base: 46
```

Essa abordagem permite que usuários com conhecimento do framework consigam rapidamente implementar análises específicas para seus próprios casos de uso.

4 Problemas e Soluções

Um dos principais desafios enfrentados durante o desenvolvimento do framework foi garantir a consistência e integridade dos dados em um ambiente de execução concorrente. Para isso, foi necessário projetar um mecanismo de comunicação entre os componentes que fosse seguro e eficiente em múltiplas threads. A solução adotada foi a implementação da classe `Buffer<T>`, que atua como o elo entre os diferentes estágios do pipeline (como *Extractors*, *Transformers* e *Loaders*).

4.1 Controle de Concorrência com Buffer

A classe `Buffer` foi desenvolvida com suporte à concorrência, garantindo que múltiplas threads possam interagir com ela de maneira segura. Isso foi alcançado por meio de três principais mecanismos:

- **Mutex:** usado para garantir exclusão mútua durante operações críticas (como `push` e `pop`), impedindo que duas threads modifiquem simultaneamente a fila interna.
- **Condition Variable:** permite que threads consumidoras fiquem em espera até que haja novos dados no buffer, evitando busy-waiting e melhorando a eficiência da execução.
- **Semáforo:** utilizado para controlar o número de elementos disponíveis, impedindo que o buffer ultrapasse sua capacidade máxima.

4.1.1 Sinalização de Término

Além da sincronização, o `Buffer` implementa mecanismos para sinalizar o término da produção de dados:

- A flag `inputTasksCreated` indica que todas as tarefas produtoras já foram instanciadas, mesmo que ainda estejam produzindo dados. Ela é ativada pelo bloco de processamento assim que ele finaliza seu envio de tarefas para a fila.
- A flag `inputDataFinished` indica que todos os dados possíveis já foram gerados e processados, e não há mais nada a ser consumido. Isso só é ativado quando não há mais nenhuma tarefa na fila a ser processada que envia dados a esse buffer, e isso é controlado tanto pelo próprio buffer, que confere o `inputTasksCreated` do bloco de entrada e quantos elementos ele contém, quanto pelo bloco de processamento de

entrada, que espera suas tarefas serem finalizadas para ativá-la (isso é feito por meio de um semáforo que controla o número de tarefas desse bloco na fila).

Essas sinalizações são fundamentais para evitar que componentes consumidores fiquem bloqueados indefinidamente esperando por dados que nunca virão.

4.2 Suporte a Múltiplos Buffers de Entrada

Durante o desenvolvimento do componente `Transformer`, foi necessário lidar com a complexidade de múltiplos buffers de entrada (`input_buffers`). O desafio central consistia em permitir que a classe processasse dados provenientes de diferentes fontes de forma coordenada, mantendo a coerência das transformações e evitando perda ou duplicação de dados.

4.2.1 Solução Implementada

A solução envolveu três etapas principais:

1. **Histórico de Dados por Origem:** Quando o número de buffers de entrada é maior que um, a classe `Transformer` aloca um vetor `historyDataframes`, onde cada posição armazena os dados acumulados de um dos buffers de entrada. A operação `hStack` é usada para empilhar horizontalmente os novos dados recebidos sobre os anteriores.
2. **Extração e Combinação de Dados:** A função `enqueue_tasks` tenta extrair um dado de qualquer um dos buffers disponíveis. Ao conseguir, ela constrói um vetor de argumentos (`args`) no qual o valor extraído é utilizado diretamente, enquanto os demais são recuperados a partir do histórico. Assim, é possível executar transformações envolvendo tanto o dado novo quanto o contexto dos demais buffers.
3. **Empacotamento e Enfileiramento da Tarefa:** Os argumentos são empacotados como ponteiros compartilhados (`std::shared_ptr<T>`) e repassados para a fila de execução. A função `create_task` é então chamada, realizando a transformação por meio da função virtual `run`.

4.2.2 Grau de Complexidade da Solução

A solução adotada apresenta um **grau de complexidade médio-alto**, tanto do ponto de vista de implementação quanto conceitual. Isso porque salvamos basicamente todos os

dados que passam pelo buffer, algo que pode aumentar muito rapidamente.

4.3 Problema ao Realizar `group_by` em um Transformer

No contexto de um pipeline de processamento paralelo, o uso do operador `group_by` apresenta desafios específicos. Os principais problemas são:

- Os dados chegam em pedaços (*chunks*) de forma assíncrona, ou seja, diversos `Dataframes` parciais são processados separadamente.
- A operação de `group_by` exige uma visão global dos dados para produzir resultados agregados corretos, o que entra em conflito com o processamento paralelo por pedaços.
- O pipeline utiliza uma fila de tarefas (`TaskQueue`) e múltiplas threads, o que implica que várias transformações ocorrem de forma concorrente.

Dessa forma, não é possível aplicar o `group_by` diretamente em cada pedaço de forma independente, pois a agregação final estaria incorreta.

4.3.1 Solução Implementada

A classe `GroupByTransformer` resolve o problema por meio de duas fases principais:

- **Agregação Parcial por Tarefa (`createAggTask`)**

  Cada entrada extraída do buffer de entrada é processada com um `group_by` local, gerando um `Dataframe` parcialmente agregado, chamado `littleAggregated`. Esse resultado é acumulado em uma estrutura global chamada `aggregated`, utilizando a função `hStackGroup()`.

  ```
  std::lock_guard<std::mutex> lock(mtx);
  aggregated.hStackGroup(littleAggregated);
  ```

  Um mutex (`mtx`) é utilizado para garantir segurança no acesso concorrente à estrutura compartilhada `aggregated`.

- **Envio em Blocos Agregados**

  Após o término de todas as tarefas de agregação parcial, o `Dataframe aggregated` é particionado em blocos (slices) e enviado aos buffers de saída. Isso garante que os dados tenham sido corretamente agrupados antes de serem propagados para as próximas etapas do pipeline.

5 Vantagens e desvantagens das escolhas

A arquitetura de execução baseada em fila de tarefas e thread pool, com threads auxiliares dedicadas e buffers de comunicação, representa uma solução interessante para o contexto proposto, mas nem tudo são flores. Esta abordagem traz consigo uma série de características que merecem análise cuidadosa no contexto dos requisitos do projeto.

5.1 Vantagens

Paralelismo eficiente: A separação entre threads auxiliares para criação de tarefas e thread pool para execução permite um aproveitamento contínuo dos recursos computacionais. Enquanto as tarefas são executadas pela thread pool, novas tarefas estão sendo preparadas simultaneamente, minimizando o tempo ocioso.

Balanceamento dinâmico de carga: A fila de tarefas atua como um mecanismo de amortecimento, permitindo que componentes com diferentes características de desempenho coexistam eficientemente. Componentes mais rápidos podem executar tarefas em ritmos diferentes de componentes mais lentos, sem que isso cause bloqueios ou gargalos no pipeline. Esta capacidade de adaptação dinâmica é especialmente relevante.

Abstrações de alto nível: A arquitetura proposta encapsula toda a complexidade de gerenciamento de concorrência, permitindo que o usuário do framework foque exclusivamente na definição dos componentes lógicos do pipeline.

Flexibilidade na composição de pipelines: A comunicação baseada em buffers de entrada/saída facilita a construção de topologias de processamento complexas e não-lineares. O sistema pode suportar naturalmente padrões como bifurcação (onde um componente alimenta múltiplos consumidores) e união (onde múltiplos produtores alimentam um único consumidor), possibilitando a representação de workflows analíticos sofisticados sem comprometer o desempenho ou a clareza conceitual.

Escalabilidade horizontal: O sistema demonstra excelente adaptabilidade a incrementos de capacidade computacional. Se mais threads forem disponibilizadas (via confi-

guração do usuário), o framework automaticamente distribuirá tarefas adicionais sem necessidade de reprojetar o pipeline.

5.2 Desvantagens

Overhead de gerenciamento: A criação e manutenção de threads auxiliares dedicadas para cada componente do pipeline representa um custo razoável de recursos do sistema. Em configurações com muitos componentes, este overhead pode ser substancial, mesmo quando não está realizando trabalho computacional intensivo, consome memória e tempo de CPU.

Complexidade de implementação: A implementação interna do framework exige um gerenciamento sofisticado de concorrência. A sincronização adequada dos buffers compartilhados, a prevenção de condições de corrida e o tratamento correto de situações de erro em um ambiente altamente paralelo podem tornar a manutenção e evolução do framework algo complicado.

Potencial desperdício em sistemas pequenos: Para cargas de trabalho modestas ou em sistemas com recursos computacionais limitados, o modelo proposto pode introduzir um overhead desproporcional aos benefícios. Em particular, tarefas muito simples podem não se beneficiar significativamente do alto grau de paralelismo, enquanto ainda precisam arcar com os custos de gerenciamento de múltiplas threads e sincronização de acessos concorrentes.

Controle de buffer: Um desafio significativo desta arquitetura é gerenciar os buffers intermediários. Para evitar o consumo excessivo de memória, é necessária a implementação de mecanismos de controle, como semáforos, que restringem a criação de novas tarefas quando os buffers atingem um limite predefinido. Esta verificação adicional introduz uma camada extra de complexidade.

Falta de priorização de tarefas: A implementação atual com uma fila única operando em regime FIFO (*First-In, First-Out*) não oferece mecanismos para diferenciar a prioridade das tarefas.

Armazenamento de memória: O sistema atualmente consome muita memória ao tentar ter `transformers` com mais de um *input* do *buffer*.

5.3 Trade-off

Apesar das desvantagens identificadas, a arquitetura proposta representa uma solução equilibrada para os desafios de computação escalável no contexto do projeto. Os benefícios de paralelismo eficiente e balanceamento dinâmico de carga são particularmente relevantes para o nosso contexto, compensando os custos adicionais de complexidade e overhead em cenários de uso apropriados. Para trabalhos futuros, o framework poderia evoluir para otimizações adicionais para reduzir o overhead em sistemas menores.

6 Projeto Exemplo

Para demonstrar a eficiência do nosso framework, vamos criar um projeto de exemplo para um sistema de reserva de viagens. Como esse é um sistema que pode precisar lidar com quantidades enormes de dados, é necessário um pipeline capaz de executar tarefas de forma paralela e eficiente, exatamente o que buscamos oferecer com nosso framework.

Para nos aproximarmos de uma aplicação real, procuramos criar análises alinhadas com o contexto do negócio e que possam gerar valor de forma efetiva.

A primeira análise está relacionada aos voos. Queremos identificar quais destinos têm a maior procura até o momento, ou seja, destinos cujos voos apresentam a maior taxa de ocupação. Isso é relevante para entender quais locais podem, possivelmente, ter um aumento no preço das passagens.

A segunda está relacionada às solicitações de reserva dos usuários e à disponibilidade geral. Queremos ter uma previsibilidade de receita e de disponibilidade futura para poder planejar nos próximos passos do negócio com mais clareza. Para tal, olhamos para a base de solicitações de reserva, agrupamos, fazemos o match com a disponibilidade e, então, considerando apenas as reservas possíveis, calculamos a receita prevista e a nova disponibilidade após as reservas. Com isso, é possível saber o quanto teremos de receita e o quanto ainda podemos vender. Essa análise é importante, também, para um possível aumento de preços ou promoções.

A figura 3 mostra uma versão diagramada do pipeline.

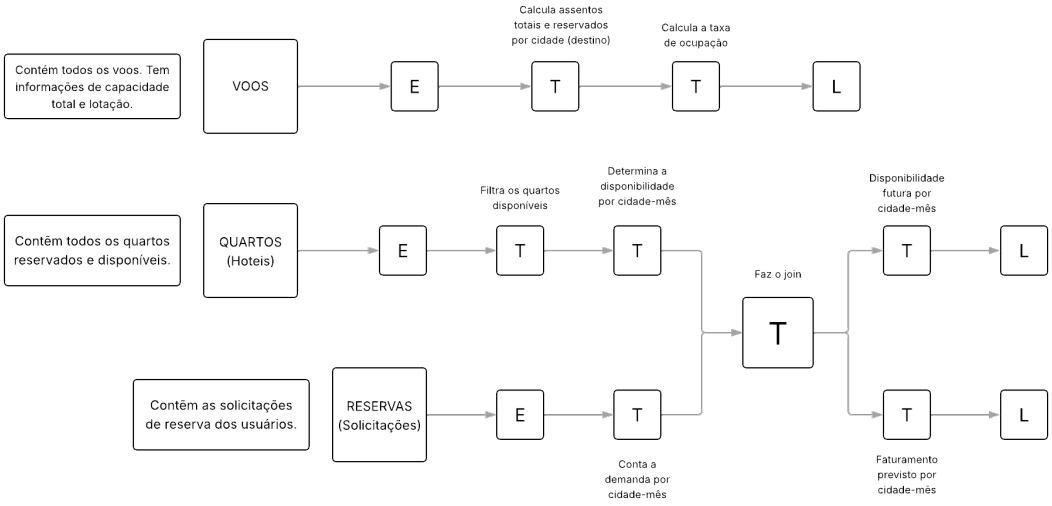

Figura 3: Pipeline de análise

Dentro desse contexto, temos duas bases sendo alimentadas como csv (VOOS e QUARTOS) e uma como um banco de dados (RESERVAS). Além disso, durante a execução do pipeline, algumas estatísticas de interesse são calculadas. Os resultados da execução desse exemplo serão mostrados, analisados e comentados na próxima seção.

7 Resultados Experimentais

Com o exemplo construído e em execução, o próximo passo é testar seu desempenho. Para isso, variaremos o número de threads e avaliaremos o tempo de execução de cada pipeline individualmente.

Os experimentos foram realizados em um computador com sistema operacional Linux Ubuntu 24.10, processador Intel i5-11300H (4 núcleos e 8 threads) e 24 GB de memória RAM. Para garantir maior confiabilidade nos resultados, cada configuração de número de threads foi testada 10 vezes, sendo registrado o valor médio de tempo de execução.

A seguir, apresentamos os resultados obtidos para o pipeline de voos (ilustrado na parte superior da Figura 3), utilizando uma base de dados de **150.000** linhas. Os tempos médios de execução estão representados no gráfico da Figura 4.

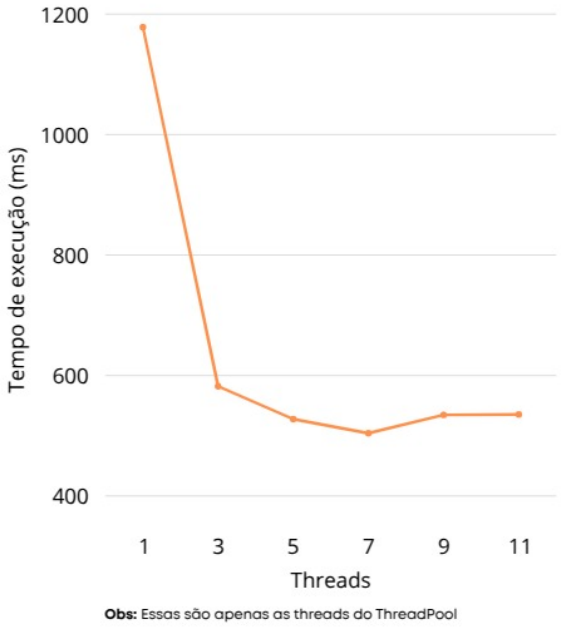

Figura 4: Experimento de tempo - Pipeline Voos

Os resultados evidenciam um ganho significativo de desempenho com o aumento no número de threads, indicando que o framework foi paralelizado com sucesso. O ganho é especialmente expressivo ao passar de 1 para 3 threads, resultando em uma redução de tempo de execução de quase três vezes. A partir de 7 threads, entretanto, a performance permanece a mesma, o que pode ser uma limitação do hardware. Ao aumentar excessivamente o número de threads, a performance tende a se degradar devido ao aumento significativo na troca de contexto entre elas.

Em seguida, apresentamos os resultados obtidos para o pipeline de hotéis e reservas, usando uma bases com **250.000** reservas e **10.000** hotéis. Os tempos médios estão representados no gráfico da Figura 5.

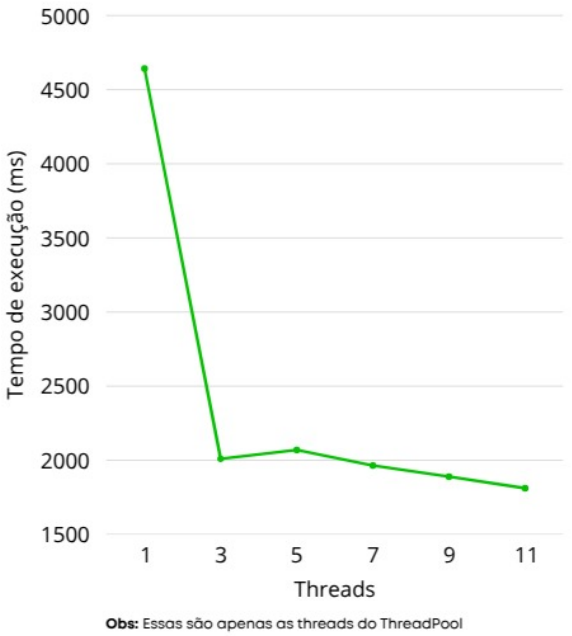

Figura 5: Experimento de tempo - Pipeline Hotéis

Vemos novamente o ganho no desempenho com a passagem do processamento com 1 thread para o com 3 threads, acompanhado por uma leve piora com 5 threads e depois uma nova melhora com o aumento das threads. Após testar os sub pipelines isoladamente, chegou a hora de juntar tudo. Temos a seguir (figura 6) o tempo de execução para cada quantidade de threads para todo o pipeline, com ambas as linhas de execução.

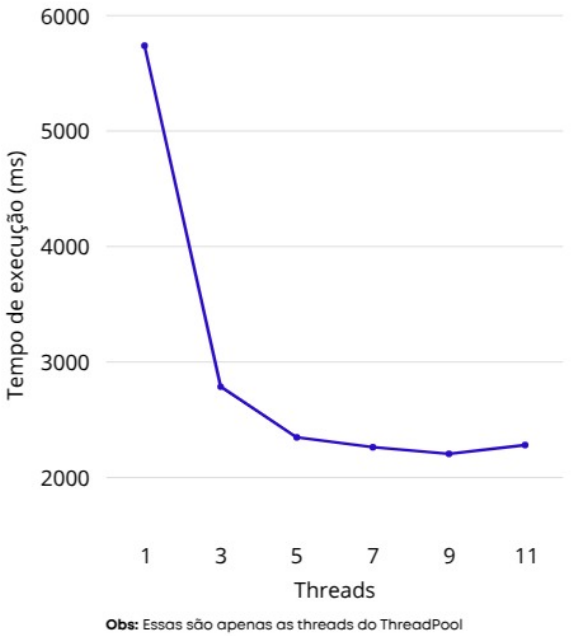

Figura 6: Experimento de tempo - Pipeline Completo

Vemos que, dessa vez, uma quantidade ainda maior de threads foi necessária para atingir o melhor desempenho permitido, provavelmente devido à maior carga de dados e de processamento solicitada. Com isso, o custo das trocas de contexto torna-se menos relevante para execuções com 3 a 5 threads do que nos casos anteriores.

Por fim, decidimos redirecionar todas as saídas (prints) do dashboard e dos loaders para um arquivo e, em seguida, refizemos os testes. O resultado obtido está apresentado na Figura 7.

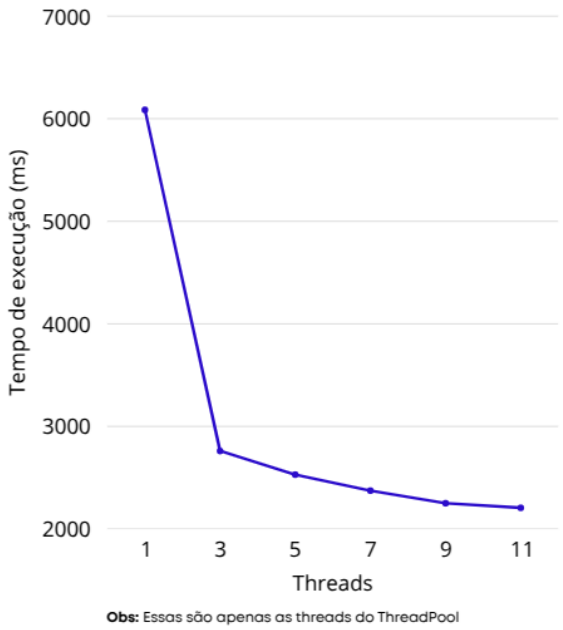

Figura 7: Experimento de tempo — Pipeline completo com loaders em arquivo

Podemos observar que não houve uma alteração significativa em relação ao gráfico anterior (Figura 6), mantendo-se o mesmo padrão de comportamento.

8 Conclusão

Ao longo deste trabalho, conseguimos projetar e implementar um framework capaz de processar grandes volumes de dados de forma eficiente e paralela. Através da combinação entre fila de tarefas, pool de threads e buffers de entrada e saída, foi possível garantir o balanceamento de carga e a execução concorrente entre os diferentes estágios do pipeline, conforme os objetivos inicialmente propostos.

Durante o processo de desenvolvimento, nos deparamos com diversos desafios práticos. Muitos deles surgiram justamente na tentativa de garantir a execução paralela de forma

eficiente. Descobrimos, na prática, que paralelizar uma aplicação é uma tarefa complexa, cheia de detalhes e armadilhas, que vão desde condições de corrida até gargalos inesperados de desempenho. Cada erro enfrentado foi uma oportunidade de aprendizado, e encontrar as soluções adequadas exigiu que revisitássemos e aplicássemos, com profundidade, os conceitos vistos em aula. No fim, após muito esforço, testes e ajustes, conseguimos superar essas dificuldades e alcançar nosso objetivo.

Os resultados experimentais obtidos demonstram claramente que o framework foi bem-sucedido em explorar os recursos computacionais disponíveis, apresentando ganhos significativos de desempenho com o aumento no número de threads utilizadas. Isso comprova, na prática, a eficácia da arquitetura desenvolvida.

Além disso, este trabalho foi essencial para consolidar nosso entendimento sobre conceitos fundamentais como concorrência, paralelismo, sincronização e balanceamento de carga, temas muitas vezes abstratos em sala de aula, mas que se tornaram concretos ao longo da construção e experimentação do sistema.

Por fim, para o futuro, vemos diversas possibilidades de evolução para o framework. Podemos aprimorar a criação de tarefas, otimizar o gerenciamento dos buffers e refinar as estruturas internas para melhorar o desempenho. Também é importante fortalecer a usabilidade, tornando o sistema mais intuitivo e prevenido a falhas de usuário, além de implementar testes mais robustos para garantir a confiabilidade. O sistema, portanto, permanece aberto para melhorias e refinamentos em projetos futuros, servindo como uma base sólida sobre a qual novas funcionalidades e otimizações poderão ser construídas.