Micro-framework data processing pipeline

Introdução

O presente relatório é referente ao trabalho de A1 passado na matéria de Computação Escalável, lecionada no 5º período do curso de Graduação em Ciência de Dados e Inteligência Artificial da FGV - EMAp. O relatório é constituído por nossa modelagem do problema proposto, passando tanto por explicações conceituais dos objetos do trabalho quanto por suas características de implementação. Ao final, também mostramos e detalhamos como é o nosso problema de exemplo, feito para testar a corretude de nossa biblioteca.

Integrantes do grupo

Anderson Gabriel Falcão dos Santos - andersonfalcaosantos@gmail.com
Guilherme Moreira Castilho - guilherme222castilho@gmail.com
Pedro Santos Tokar - pedrotokar2004@gmail.com
Tomás Paiva de Lira - tomaspaivadelira1@gmail.com
Vitor Matheus do Nascimento Moreira - vitor.mnw@gmail.com

Dataframe

Em um primeiro momento, a modelagem do Data Frame foi feita a partir da integração de duas classes: a classe **`BaseColumn`**, que representa os vetores-coluna, e a classe **`DataFrame`**, cuja função principal é aglutinar cada uma das **`BaseColumn.`**

Cada coluna da tabela possui as seguintes propriedades:

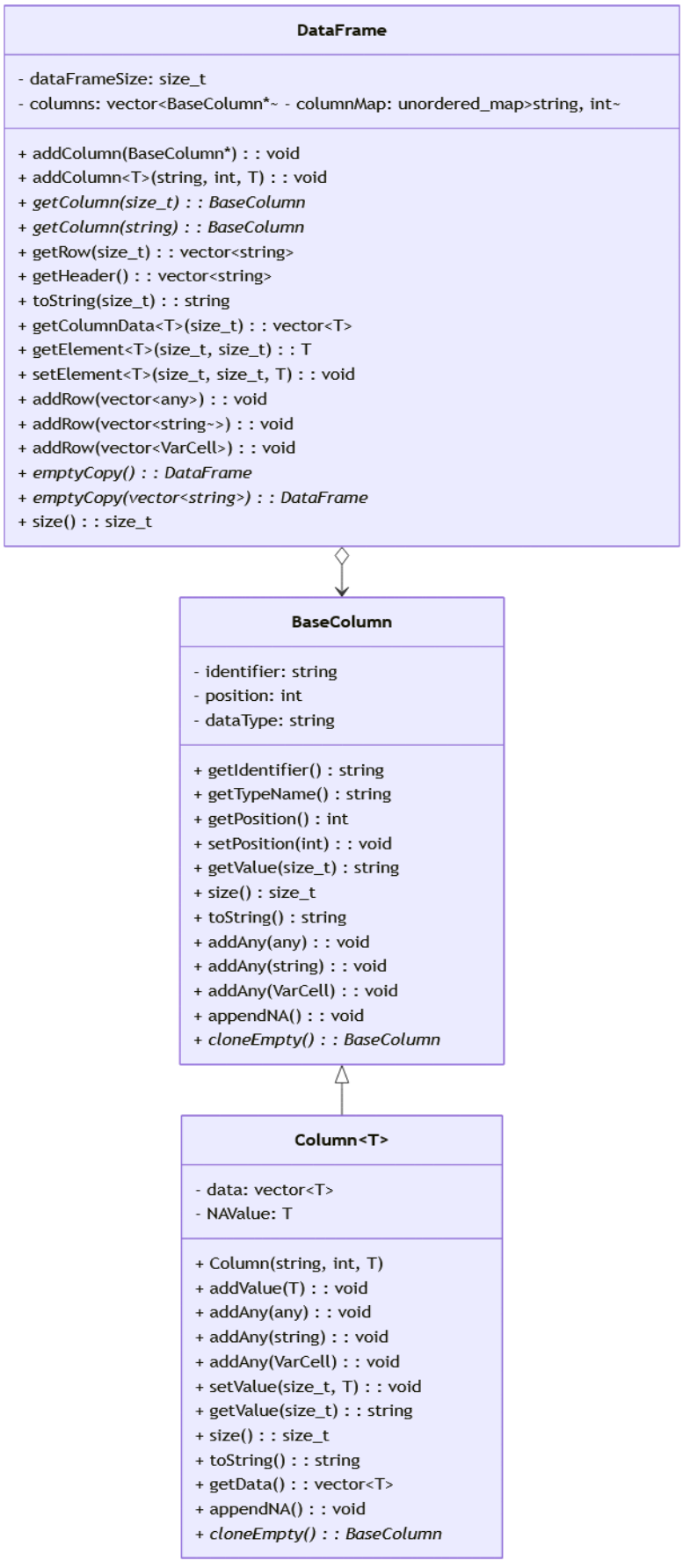
- Identificador
- Posição no DataFrame
- Tipo de Dado Explícito (Homogeneidade das Séries)
- Um **`std::vector`** que guarda os elementos do tipo especificado
- Utiliza um valor pré-fixado para representar o **`N.A`**, isso varia para cada tipo de dado, e pode ser alterado pelo usuário na hora da criação de uma coluna (é uma propriedade da coluna e não do dataframe). Caso o usuário não deseje alterar o valor nulo, é utilizado um valor padrão definido em **`types.h`**.

Usamos uma classe abstrata para representar as colunas e outra classe que a herda. Assim, conseguimos abranger diferentes tipos de dados de forma uniforme e não precisamos conhecer o tipo específico para definir a manipulações. Herdamos a classe base utilizando uma Template para o tipo, assim conseguimos garantir a homogeneidade dos dados e também podemos guardar todos os ponteiros das Columns sem a necessidade de saber o seu tipo.

Essa modelagem levou em consideração a sua interação com os tratadores, pois, por meio do uso dos atributos das séries, conseguimos verificar se as operações almejadas estão devidamente definidas.

Em resumo, o Data Frame nada mais é que uma coleção de colunas abstratas, com as seguintes capacidades:

- Adicionar Colunas
- Acessar colunas e linhas especificadas (utilizando índices ou nome)
- Cópia da estrutura do DataFrame (nomes e tipos das colunas).
- Exibir a tabela (através de um output no terminal).
- Uso do `<shared_ptr<BaseColumn>>`, permitindo a manipulação para funções não definidas no Data Frame (Múltiplos Tratadores).
- Adicionar novas linhas de forma genérica (utilizando vectors de variant, any ou string)


DataFrame
- dataFrameSize: size_t
- columns: vector<BaseColumn*~ - columnMap: unordered_map>string, int~
+ addColumn(BaseColumn*) : : void
+ addColumn<T>(string, int, T) : : void
+ getColumn(size_t) : : BaseColumn
+ getColumn(string) : : BaseColumn
+ getRow(size_t) : : vector<string>
+ getHeader() : : vector<string>
+ toString(size_t) : : string
+ getColumnData<T>(size_t) : : vector<T>
+ getElement<T>(size_t, size_t) : : T
+ setElement<T>(size_t, size_t, T) : : void
+ addRow(vector<any>) : : void
+ addRow(vector<string~>) : : void
+ addRow(vector<VarCell>) : : void
+ emptyCopy() : : DataFrame
+ emptyCopy(vector<string>) : : DataFrame
+ size() : : size_t
BaseColumn
- identifier: string
- position: int
- dataType: string
+ getIdentifier() : string
+ getTypeName() : string
+ getPosition() : int
+ setPosition(int) : : void
+ getValue(size_t) : string
+ size() : size_t
+ toString() : string
+ addAny(any) : : void
+ addAny(string) : : void
+ addAny(VarCell) : : void
+ appendNA() : : void
+ cloneEmpty() : : BaseColumn
Column<T>
- data: vector<T>
- NAValue: T
+ Column(string, int, T)
+ addValue(T) : : void
+ addAny(any) : : void
+ addAny(string) : : void
+ addAny(VarCell) : : void
+ setValue(size_t, T) : : void
+ getValue(size_t) : : string
+ size() : : size_t
+ toString() : : string
+ getData() : : vector<T>
+ appendNA() : : void
+ cloneEmpty() : : BaseColumn

Repositório de dados

São as partes do projeto responsáveis por se comunicarem (conectarem) diretamente com os dados a serem manipulados. Uma vez que estaremos trabalhando com repositórios tabulares, projetamos esse componente para realizar operações por linha da tabela.

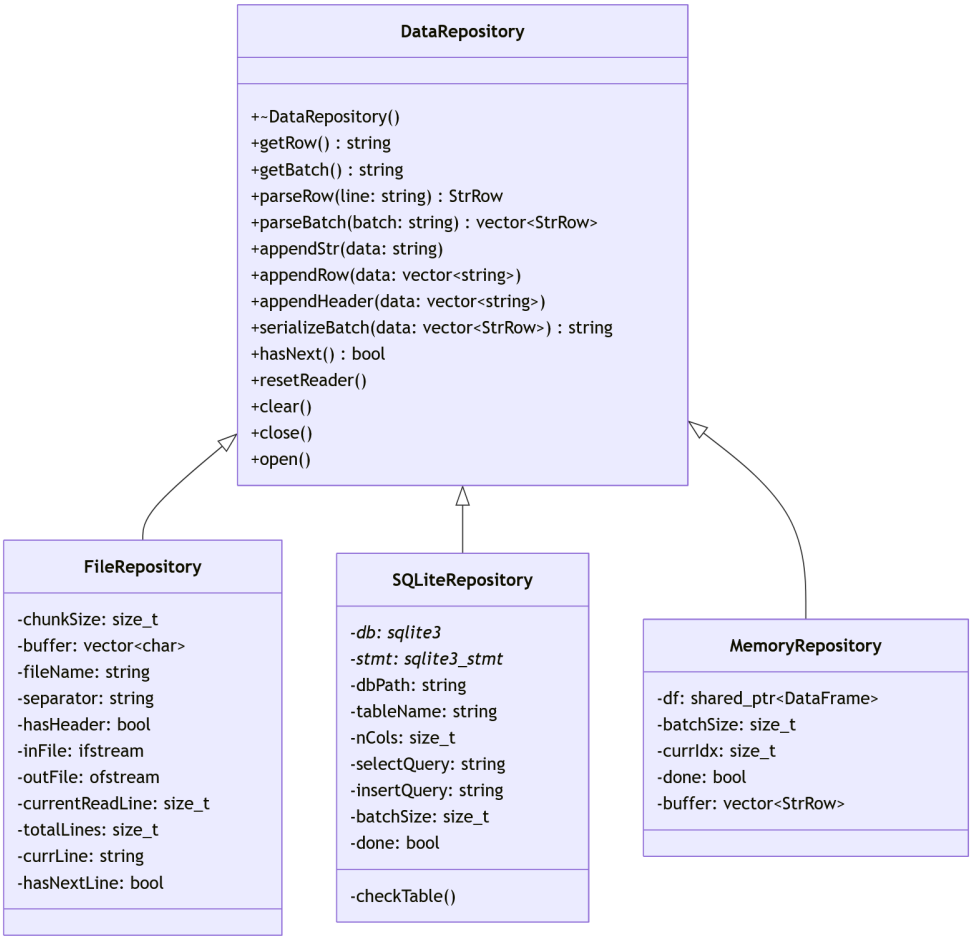
O que fizemos foi criar uma classe abstrata **`DataRespository`**, que representa tudo que um repositório de dados específico precisa implementar para ser integrado aos outros componentes do framework de forma genérica (Extrator e Loader).

O framework já fornece por padrão três classes específicas, herdadas da **`DataRepository`**: a **`SQLiteRepository`**, **`FileRepository`** e **`MemoryRepository`**. Cada uma delas tem um construtor apropriado para o contexto (por exemplo, a de SQL recebe informações para conectar ao banco de dados e a tabela que deve ser utilizada, enquanto a de arquivo recebe o caminho do arquivo e o separador). Não são realizadas inferências em relação aos tipos de dados, e é tarefa do usuário garantir que os tipos e nomes das colunas estejam alinhados com cada DataFrame que ele quiser utilizar.

Informações de contexto importantes para cada:

- **`SQLiteRepository`:** conexão, query básica para criação de novas tabelas
    - Ao instanciar essa classe, abrimos um .db já existente ou criamos um. Depois é necessário abrir uma tabela já existente ou criar uma tabela nova no .db através de uma query.
- **`FileRepository`:** caminho do arquivo, informação de separador.
    - Recebe o caminho do arquivo e o separador. Funciona para arquivos de dados tabulares separados por uma string ou char.
- **`MemoryRepository`:** endereço de memória.
    - Esse repositório faz a conexão com um DataFrame já existente em memória, recebendo um **`shared_ptr`** para o DataFrame. É útil no contexto em que outras partes de um programa gerem um DataFrame que precise entrar na pipeline futuramente.

Tais classes são posteriormente associadas aos Extratores e Loaders através de referências.



Tratadores

São as partes do framework que cuidam das operações da pipeline, ou seja, da manipulação dos dataframes.

Essencialmente, **os tratadores são blocos de construção do ETL** que recebem um ou mais dataframes, realizam operações baseando-se nos elementos deles e salvam os resultados em um ou mais dataframes com os formatos de saída especificados do tratador. Ou seja, eles realizam operações em cima de dataframes.

Associações entre tratadores e transmissão de dados

Os tratadores, por serem blocos de construção do ETL, são associados a outros tratadores tanto por entrada (etapas anteriores do ETL) quanto por saída (etapas posteriores do ETL). Com essa associação, cria-se uma relação de dependência: um tratador só consegue operar quando os anteriores já tiverem terminado suas operações. Como uma ETL pode depender de várias operações em paralelo, cada tratador pode ter grau de entrada e saída maior do que 1.

O fluxo dos dados é definido em conjunto com os tratadores: **cada tratador tem associado a si mesmo uma ou mais especificações de dataframes de saída**, que determinam como é o formato do resultado final da operação daquele dataframe (ou seja, quantas e quais colunas os dataframes com os resultados da operação do tratador terão). Os tratadores subsequentes terão acesso ao seu resultado por meio de uma referência ao seu dataframe e os índices dele (elaborado na próxima sessão).

Logo, se um tratador tem grau de entrada maior do que 1, ele terá acesso aos dataframes de todos os anteriores. Todos os seus tratadores subsequentes terão acesso ao seu dataframe para ler. O início e o fim da pipeline são definidos pelos Extratores e Loaders, que são explicados mais à frente.

Interface de operações e paralelização

Assim como explicado anteriormente, um tratador, quando for operar, terá acesso aos dataframes computados pelos tratadores que estão associados por trás com ele. Logo, a essência de funcionamento de um tratador é executar uma função que tem como entradas:

- Tuplas contendo referências para os dataframes resultados das etapas anteriores e seus índices
- Referências aos dataframes em que deverão ser escritos os resultados.

Essa função definirá qual operação o dataframe realiza, e ela deverá adicionar os resultados nos dataframes de saída do tratador.

Serão disponibilizados dois tipos de tratadores: os paralelizáveis e os não-paralelizáveis. Isso será definido por uma flag do tratador, que dirá ao orquestrador como ele deverá tratar essa etapa do ETL.

Nos tratadores não-paralelizáveis, essa função é executada quando os tratadores anteriores terminam suas tarefas, e a execução é simples: a função é chamada por uma única linha de processamento, e os índices de entrada dos dataframes anteriores serão passados por completo para a função.

Nos tratadores paralelizáveis, a função irá receber apenas parte dos índices dos dataframes de entrada, que indicam para a função em quais linhas dos dataframes ela deve operar e levar em conta para obter os resultados. A ideia por trás disso é que diferentes execuções da função sejam informadas sobre quais conjuntos de linhas (blocos) do dataframe elas devem operar.

A divisão das linhas que vão para cada execução (thread) é controlada pelo orquestrador da pipeline e é feita para que as threads recebam “pedaços” iguais dos dataframes anteriores para processar. **Ou seja, a biblioteca fornece ao usuário uma forma de programar suas funções de forma paralela sem se preocupar com algumas das dificuldades associadas a isso.** Através de parâmetros no momento em que se define os fluxos dos tratadores é possível definir configurações mais complexas, do tipo “Os índices de um dataframe são particionados em blocos e os índices do outro não”, a depender da necessidade do usuário (existem

operações em que a divisão de apenas um dos dataframes é interessante, e se deseja ter acesso total aos outros, como operações de left join).

Mesmo nos tratadores paralelizáveis, a escrita dos resultados é feita nos dataframes de saída, que são compartilhados entre execuções paralelas. Logo, é responsabilidade do usuário ter consciência de que outras threads podem escrever no dataframe, e programar suas operações de acordo (exemplo: em tratadores que fazem agregações e contagens, é necessário verificar se resultados parciais já foram inseridos no dataframe de saída).

Tratadores, extratores e loaders

Assim como citado, os tratadores são blocos de construção da ETL. Existem, então, duas categorias especiais de tratadores, que são o começo e o final da pipeline.

Extratores

Os extratores são o primeiro componente do fluxo ETL. Eles são responsáveis por extrair os dados através de algum conector do Repositório de Dados recebido como atributo, e o fazem usando métodos diversos fornecidos pelos Repositórios de Dados, escrevendo os dados lidos em um dataframe de saída, que será disponibilizado para os tratadores em sua frente. Esse tratador pode ou não ser paralelizável, a depender da natureza do repositório de dados associado a ele. Existem repositórios que se beneficiam de paralelismo e existem repositórios que não se beneficiam.

A execução do extrator sem paralelizar funciona pegando linha por linha do repositório de dados, fazendo o parser dessa linha e adicionando a linha ao DataFrame de saída. Já no caso MultiThread, utilizamos o padrão do produtor consumidor (mas com com apenas um produtor); o produtor é responsável por fazer a leitura do arquivo em lotes e adicionar esses lotes a um buffer compartilhado, depois as Threads consumidoras retiram ou esperam por elementos do buffer para trabalhar (fazendo o parser e a inserção no DataFrame), as operações de retirar e inserir elementos no buffer são protegidas por um mutex, e a inserção no DataFrame também é protegida por um mutex.

Loaders

Os loaders são semelhantes aos extratores, mas para o final do ETL: no lugar de terem tratadores posteriores a eles, eles inserem linhas em algum repositório de dados.

A lógica de inserção no repositório de dados no caso single Thread é inserir uma linha do DataFrame no repositório de dados a cada iteração. No caso MultiThread, dividimos o DataFrame de entrada do Loader em blocos (conforme o número de Threads disponibilizadas) e é feita a serialização do bloco para inserção no repositório de dados. A inserção no repositório é protegida por um mutex.

Implementação

A implementação de nossa biblioteca dos blocos de construção do ETL é feita utilizando uma classe abstrata `Task` e três classes herdadas delas (e como explicado abaixo, mais algumas subclasses): a `Transformer`, a `Extractor` e a `Loader`.

`Task`

A classe abstrata contém a implementação de funções e atributos que são comuns a todos os blocos. O que há nela é dividido em algumas categorias:

- Métodos responsáveis pelo gerenciamento dos relacionamentos entre blocos, que incluem interfaces para o usuário construir o grafo (`addNext`);
- Métodos para definir opções do bloco (como se ele deve ou não ser paralelizado) e definir estimativas de complexidade dos transformadores;
- Métodos que servem para declarar como são os dataframes de saída da Task em questão;
- Métodos que administram recursos internos (exemplos: contador de depências de tasks anteriores) que são consumidas pela linha de execução que administra o andamento da pipeline. Isso inclui métodos que realizam a limpeza de dataframes que já não estão mais sendo consumidos.
- Métodos abstratos que definem interfaces comuns de execução para as subclasses. Como será explicado adiante, o processo que orquestra a pipeline precisa ser capaz de iniciar a execução, tanto simples quanto paralela, dos blocos do ETL. Por isso, é necessário que a `Task` defina métodos abstratos (como `executeMonoThreading` e `executeMultiThreading`) que são respeitados pelas três classes herdadas.

Esses dois últimos métodos são sobrescritos por cada uma das três classes filhas descritas abaixo. O primeiro simplesmente executa normalmente a função do bloco, enquanto o segundo cuida do gerenciamento de threads e lógicas específicas para paralelização. Ele é feito de uma maneira que, quando chamado, retorna um vetor de threads e um vetor de variáveis booleanas, que são atualizadas pelas threads que terminam seus trabalhos. Dessa forma, **não é necessário ter mais que uma thread que fica em estado de espera,** já que os vetores de booleanos permitem que apenas uma thread que orquestra a pipeline verifique o andamento de várias threads diferentes.

O usuário não interage diretamente em nenhum momento com a classe Task, mas sim com suas três classes herdadas. Ainda assim, métodos usados em todas elas são definidas nessa classe.

`Transformer`

A classe `Transformer` é herdada da classe `Task`, e representa blocos que não salvam nem leem arquivos, ou seja, blocos que operam em dataframes de blocos anteriores e disponibilizam dataframes tratados para novos blocos. Esses blocos são personalizáveis pelo usuário: um usuário deve programar seus próprios `Transformers` que geram a operação desejada por ele.

A interface para o usuário programar uma operação de `Transformer` é feita por meio de herança: o usuário deve criar uma classe que herda da `Transformer` e sobrescrever um método `transform`.
A entrada desse método é composta de:

- Um vetor contendo ponteiros para os dataframes que deverão ser populados com a saída desse tratador. Eles são do formato que o próprio usuário define como saída do transformador, usando métodos implementados na classe pai Task;
- Pares de (índices, ponteiros) representando os dataframes de entrada do tratador e em quais índices de cada um ele deve operar. Isso permite com que a função seja implementada pensando em diferentes threads a executando.

Devido a natureza das operações poderem ser paralelas, ao fazer uma subclasse e sobrescrever o método abstrato `transform`, o usuário deve se atentar para a criação de `Mutex` que protejam corretamente o acesso ao DF de saída do bloco.
Internamente, a classe `Transformer` implementou alguns métodos abstratos da classe `Task` que fazem uma mediação entre chamadas que o orquestrador faz para executar o bloco e as chamadas que efetivamente são feitas para o método `transform` do usuário, cuidando de questões de chamadas de thread e divisão de índices. O usuário não precisa se preocupar com criar threads para executar sua operação, já que métodos internos fazem isso.

Extractor

A classe Extractor é também herdada da `Task`, logo possuindo a maioria dos métodos dela. Ao contrário da classe `Transformer`, essa não opera com dataframes de etapas anteriores, e sim com instâncias da classe `DataRepository`. Para cada tipo de repositório de dados há uma classe herdada específica de Extractor: `ExtractorFile`, `ExtractorSQLite` e `ExtractorMemory`, que implementam os métodos abstratos da Task de acordo com as necessidades de cada tipo de repositório.

Ao contrário da `Transformer`, o usuário não precisa criar uma subclasse para utilizar os extratores: basta instanciar algum dos extratores específicos. Dentre os extratores, não há diferenciação na forma de utilizar os métodos — as alterações na lógica de extração e inserção são feitas internamente apenas — nem na lógica de paralelização explicada acima.

Outra diferença para os transformadores é que a saída é única, ou seja, é gerado apenas um DataFrame por extrator.

Loader

Assim como `Extractor` e `Transformer`, `Loader` também herda da `Task`, e segue uma ideia muito próxima do extrator, porém no caminho contrário: recebe um único DataFrame de entrada e o “passa” para um `DataRepository` —Segue a mesma lógica do extrator novamente, tendo as subclasses `LoaderFile`, `LoaderSQLite` e `LoaderMemory`, onde cada

uma é ligado há uma subclasse de `DataRepository`. Os repositórios de dados podem processar as informações “passadas” de 2 formas:

- Adicionando uma linha ao final do repositório; ou
- Criando um/Sobrescrevendo o repositório.

O que diferencia essas duas formas é o parametro booleano `clearRepository`, que o `Loader` recebe ao instanciar a classe, já que uma vez que você limpa um repositório, basta adicionar linhas ao final para sobrescrever (Se o arquivo não existe, ele é criado automaticamente).

Para garantir o funcionamento do ETL com o Loader recebendo um único DataFrame de entrada, ao inicializar a classe há um parametro `inputIndex` que indica qual DataFrame da saída do tratador anterior a ele será carregado.

Triggers

Os triggers são representados por uma classe abstrata, **`Trigger`**. A biblioteca fornece duas classes de Trigger já implementadas: **`TimerTrigger`** e **`RequestTrigger`**. Para que um trigger seja capaz de executar corretamente a pipeline, ele deve ser informado de todos os extratores dela. Dessa maneira, o trigger consegue percorrer corretamente o DAG gerado pelas tarefas e pelas dependências (ordem de execução das tarefas).

Ambas tem um método **`start`**, que executa a pipeline uma vez, no caso do **`RequestTrigger`**, ou várias vezes, no caso do **`TimerTrigger`**. Esse método recebe um número fixo de threads, que é escolhido pelo usuário e que é usado para todos os cálculos e alocações como um limite superior..

Para efetivamente executar a pipeline, o trigger usa um método interno que implementa a lógica de orquestração das tarefas e gerenciamento das threads.

Durante a execução da pipeline, as threads são alocadas para cada tratador, fazem suas tarefas, e após terminarem, são alocadas threads para os próximos passos.

O usuário será capaz de fazer uma classe herdada da trigger para implementar a lógica de início que for necessária para ele. O método interno que orquestra efetivamente a pipeline sempre será o mesmo, e deverá ser chamado pelo usuário em sua lógica de uso do **`Trigger`**.

Orquestrador

A pipeline é composta por objetos *Task*, interligados por dependências. Cada *Task* pode ter predecessores e sucessores, o que permite a construção de grafos direcionados acíclicos (DAGs).

O orquestrador é o método da classe Trigger, responsável por gerenciar a execução das tarefas.

Heurística de pré-balanceamento de carga:

1. O usuário recebe a responsabilidade de estimar o peso individual de cada task na pipeline, levando em conta o custo de executar a mesma;
2. O *algoritmo otimizador* calcula o custo esperado de cada *task*, atribuindo esse custo a cada vértice. Com isso temos um DAG onde cada vértice representa uma *task* e cada aresta representa o custo da transição;
3. O orquestrador utiliza essa informação dos pesos para definir a ordem de execução das *task*, e calcular a quantidade de threads utilizadas em cada *task* **durante o tempo de execução,** levando em conta fatores como a fila de tarefas e a quantidade de threads disponíveis.

O algoritmo otimizador utiliza as seguintes heurísticas:

$$W^i_{\text{final}} := \alpha \cdot W^i_{\text{crit}} + (1-\alpha) \cdot \frac{W^i_{\text{sum}}}{Child_i + 1}$$

$$W^i_{\text{crit}} := W^i_{\text{base}} + \max_{\text{para cada } j \text{ filho de i}} W^j_{\text{crit}}$$

$$W^i_{\text{sum}} := W^i_{\text{base}} + \sum_{\text{para cada } j \text{ filho de i}} W^j_{\text{sum}}$$

$$Child_i := \text{Quantidade de filhos do vertice i}$$

Intuição:

- W_crit: o caminho mais custoso
- W_sum: o custo total da subárvore que parte desse vértice.
- W_final: um média ponderada de W_crit e W_sum normalizado pelo número de vértices na subárvore.

O hiperparâmetro *alpha* é escolhido pelo usuário levando em conta as ‘peculiaridades’ da pipeline e realizando testes.

Dessa maneira, esperamos calcular, a priori, informações relevantes que levem a uma distribuição de *threads* ótima, minimizando o tempo total de processamento da *pipeline*.

Detalhamento do método que calcula os pesos:

O método *calculateThreadsDistribution* tem como objetivo calcular pesos heurísticos para cada tarefa da pipeline, a fim de permitir uma alocação mais eficiente e proporcional de threads durante a execução paralela. Esses pesos são usados posteriormente para priorizar tarefas e definir quantos recursos computacionais devem ser destinados a cada uma delas, levando em conta tanto o caminho crítico quanto a carga total descendente.

O funcionamento do método é dividido em duas fases principais:

1. Travessia Direta (BFS 1) – Construção do Grafo e Cálculo de Níveis

Nesta etapa, realiza-se uma travessia em largura (BFS) partindo dos *extractors* (tarefas iniciais da pipeline). O objetivo é:

- Construir o *taskMap*, um mapa que associa nomes de tarefas a objetos *taskNode*.
- Atribuir o nível de profundidade *level* de cada tarefa, relativo à distância dos *extractors*.
- Identificar as tarefas finais (aquelas sem sucessores), que serão usadas como ponto de partida na fase inversa.

Esse processo garante que o grafo da pipeline está completo e pronto para o cálculo de pesos.

2. Travessia Reversa (BFS 2) – Cálculo de Pesos

Partindo das tarefas finais (possíveis *loaders*), realiza-se uma segunda travessia em ordem reversa. Para cada *taskNode*, são computadas três métricas principais:

- *cpWeight* (critical path weight): soma do peso base da tarefa com o maior peso de caminho de seus filhos. Representa o custo do caminho mais longo até uma folha.
- *sumWeight*: soma total dos pesos de todas as tarefas descendentes.
- *numChild*: número de tarefas descendentes.

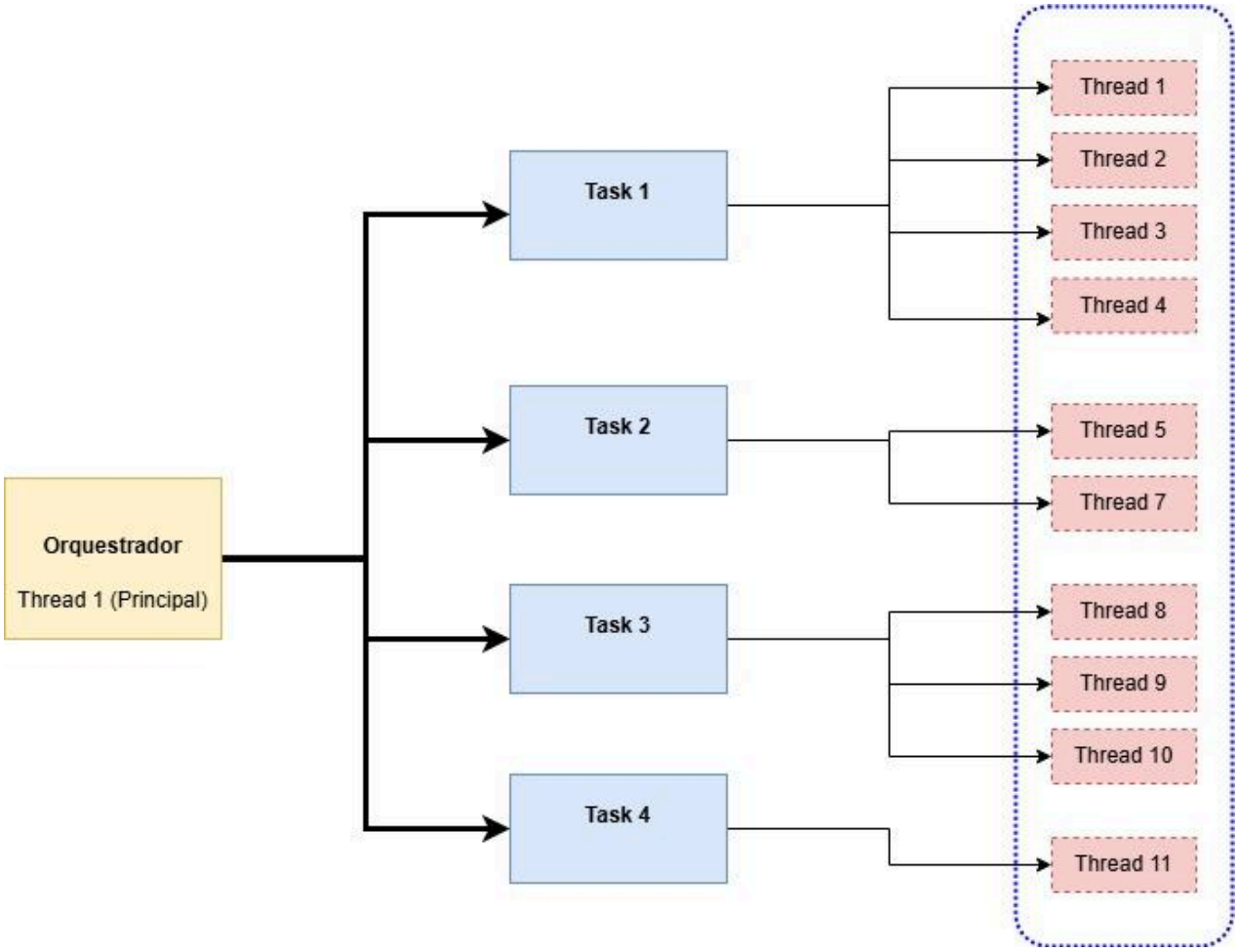
A heurística de peso final (*finalWeight*) é calculada com a fórmula que já foi mostrada acima.

Esse valor balanceia entre dar prioridade para tarefas no caminho crítico e considerar a carga total esperada da subárvore de execução. O uso de alpha permite controlar esse equilíbrio (com valor padrão 2/3).

Este método é essencial para garantir que a orquestração paralela seja guiada por uma estratégia inteligente de distribuição de threads, evitando tanto o subaproveitamento de recursos quanto sobrecarga em tarefas mal balanceadas

Detalhamento do método orquestrador:

O método *orchestratePipelineMultiThread* utiliza uma combinação de balanceamento heurístico e controle reativo para coordenar a execução eficiente de tarefas em múltiplas threads. Sua principal função é decidir dinamicamente quais tarefas devem ser executadas, quantas threads cada uma poderá utilizar, e como sincronizar os grupos de execução em tempo de execução.



1. Preparação e Priorização

A função começa invocando *calculateThreadsDistribution(maxThreads)* para calcular os pesos heurísticos (`finalWeight`) de cada tarefa com base na estrutura da pipeline.

Em seguida, é construído um *std::set* chamado *tasksQueue*, que ordena os nomes das tarefas com base em seu *finalWeight*. O comparador utilizado garante que tarefas mais pesadas (prioritárias) fiquem no topo da fila, e desempata pelo nome para garantir ordem determinística.

2. Alocação Dinâmica de Threads

Enquanto houver tarefas prontas e threads disponíveis, o método extrai a tarefa de maior prioridade da *tasksQueue* e decide quantas threads alocar para sua execução. A lógica de decisão leva em conta:

- Se a tarefa pode ser paralelizada (*canBeParallel()*).
- A proporção máxima de threads que a tarefa aceita (getMaxThreadsProportion()).
- O peso relativo da próxima tarefa na fila (para evitar que uma tarefa consuma todos os recursos).

Esse cálculo assegura que a distribuição de threads seja proporcional à carga estimada de cada tarefa, otimizando o uso do paralelismo disponível.

3. Execução em Grupo: *ExecGroup*

Cada tarefa é executada como um *ExecGroup*, que contém:

- A tarefa (*task*).
- Um vetor compartilhado de flags atômicas (*flags*) que indicam a conclusão de cada thread.
- A lista de threads criadas.
- Um vetor *joined* que rastreia quais threads já foram sincronizadas.

As threads são disparadas via *executeMultiThread*, e cada uma é responsável por atualizar sua flag ao final da execução e sinalizar o orquestrador através da *condition_variable*.

4. Monitoramento e Sincronização

O orquestrador monitora constantemente os *ExecGroups* ativos:

- Quando uma flag indica que uma thread terminou, ele chama *join()* nessa thread e libera o recurso.
- Quando todas as threads de um grupo terminam, a tarefa é marcada como concluída via *finishExecution()*, e suas tarefas sucessoras (caso estejam prontas) são adicionadas à *tasksQueue*.

Se não houver tarefas prontas e o sistema estiver aguardando a conclusão de threads, o orquestrador entra em modo de espera usando *orchestratorCv.wait(...)*, sendo notificado assim que qualquer thread terminar.

5. Benefícios e Eficiência

Essa abordagem apresenta diversas vantagens:

- Alta eficiência no uso de threads, evitando ociosidade e sobrecarga.
- Reatividade, já que o orquestrador é notificado assim que *threads* terminam.

- Balanceamento dinâmico de carga, com alocação proporcional e baseada em heurística.
- Gerenciamento ordenado, com sincronização adequada e respeitando dependências entre tarefas.

6. Desvantagens da Abordagem

Apesar de eficiente, a abordagem apresenta limitações importantes:

- Alta complexidade: o uso combinado de mutexes, *condition_variable*, threads manuais e flags atômicas torna o código difícil de manter e propenso a erros como *race conditions* ou *deadlocks*.
- Sobrecarga de sincronização: em pipelines com tarefas leves, o custo de coordenação pode superar o benefício do paralelismo.
- Alocação subótima de threads: a heurística pode ser conservadora ou imprecisa, dependendo da qualidade das estimativas (*finalWeight*, *getBaseWeight*), levando à subutilização de recursos.
- Sem realocação dinâmica: uma vez iniciada, uma tarefa mantém suas *threads* até o fim, mesmo que outras tarefas prioritárias estejam esperando.
- Maior uso de memória: cada grupo ativo mantém estruturas auxiliares (*threads*, *flags*, *status*), o que pode ser custoso em pipelines muito grandes.

7. Próximos passos

A abordagem implementada pode ser estendida ajustando dinamicamente os pesos inicialmente atribuídos pelo usuário às tarefas, com base no desempenho real observado durante a execução. Em vez de depender exclusivamente dos valores estimados como *baseWeight* e *pThreads*, o sistema poderia monitorar métricas como tempo de execução, uso de CPU ou número de threads ociosas para recalibrar os pesos das tarefas em tempo real. Isso permitiria uma distribuição de threads mais precisa e eficiente, especialmente em pipelines onde a carga de trabalho varia de forma imprevisível ou os pesos fornecidos inicialmente não refletem bem o custo computacional das tarefas.

Detalhando os hiperparâmetros:

O usuário tem um papel importante na configuração da pipeline, sendo responsável por definir três hiperparâmetros principais: *baseWeight*, *pThreads* e *alpha*.
O *baseWeight* representa uma estimativa do custo computacional de cada tarefa e influencia diretamente o cálculo do peso final utilizado na distribuição de threads; se mal estimado, pode levar a alocação ineficiente de recursos.
O *pThreads* (proporção máxima de threads) define o limite superior de threads que uma tarefa pode utilizar, controlando o grau de paralelismo permitido por tarefa; valores altos podem monopolizar recursos, enquanto valores baixos podem subaproveitar a capacidade do sistema.
Já o *alpha* é um fator de ponderação que equilibra a importância do caminho crítico (*cpWeight*) e da soma total de cargas descendentes (*sumWeight*) no cálculo do *finalWeight*, valores

próximos de 1 favorecem tarefas no caminho crítico, enquanto valores menores priorizam a carga acumulada. A escolha adequada desses hiperparâmetros impacta diretamente a eficiência e o balanceamento da execução paralela.

Projeto de Exemplo

Sistema para aprovação de transações financeiras

Objetivo

O Banco **VT.PGA** recebe um grande fluxo de transações financeiras por segundo. O banco **VT.PGA** fornece uma proteção robusta contra fraudes, e precisa de utilizar nosso framework para garantir que as transações a aprovação de transações financeiras seja rápida e confiável.

> Queremos criar um sistema que avalia em tempo real se uma transação financeira poderá ser liberada automaticamente ou bloqueada e encaminhada para uma outra análise.
>
> Ou seja, queremos verificar se determinadas transações que ocorreram em um intervalo de tempo vão ser liberadas automaticamente ou não (ao final, salvar em um banco de dados se a transação foi liberada ou não).

A forma que uma transação é verificada para liberação envolve duas operações principais:

- **Cálculo determinístico:** Verifica se o saldo e limite do cliente são compatíveis com o valor da transação.
- **Cálculo de risco:** Calcula valores de risco para características que podem indicar se uma transação é fraudulenta. Se uma certa função dos valores de risco for maior que um certo limiar a transação não é liberada.

Início da pipeline: A pipeline se inicia através de um trigger de requisição (o banco agrupa N transações que ocorreram em um certo intervalo de tempo e requisita o início da pipeline de liberação das transações).

Mocks

Esse script em Python é responsável por gerar dados simulados (mock) para três tabelas principais usadas em uma análise de transações financeiras: **regiões**, **usuários** e **transações**.

Primeiro, ele cria uma tabela com informações regionais regioes_estados_brasil.csv, atribuindo a cada estado brasileiro uma latitude, longitude, média transacional mensal e número de fraudes recentes. Em seguida, para o tamanho definido, ele gera dados fictícios de usuários, cada um com um id_usuário único, saldo, estado de origem e limites de pagamento por modalidade (PIX, TED, DOC, Boleto). Os dados são salvos em arquivos .csv e também inseridos em um banco SQLite.

Depois disso, o script gera uma tabela de transações em que cada linha representa uma transação entre dois usuários, com dados como modalidade, valor, data e região. Por fim, ele realiza uma verificação simples para identificar quantas dessas transações envolvem um pagador cujo saldo é insuficiente para cobrir o valor da transação. Também atribui a cada usuário um saldo simulado com distribuição exponencial, representando sua conta bancária.

A análise estatística de cada arquivo resultante da pipeline de saída foi realizada no noteboo summary_analysis.ipynb.Concluímos que, após os tratamentos aplicados, a plotagem dos histogramas revelou valores coerentes para a variável *score*. Além disso, ao analisar a variável *saldo*, observamos que a taxa de aprovações e reprovações condiz não apenas com as distribuições uniformes e exponenciais modeladas no mock, mas também confirma que o processo avaliativo está sendo conduzido de forma adequada e consistente.

Análises

Análise 1: Aprovações

Análise feita sobre o objetivo final da pipeline: analisar as transações que foram (ou não) aprovadas.

Análise 2: Relação Risco Valor X Aprovação

Análise que consiste em verificar as relações entre o score de risco de valor (será apresentado mais à frente) e o fato da aprovação (ou não).

Análise 3: Relação Risco Horário X Aprovação

Análise que consiste em verificar as relações entre o score de risco de horário e o fato da aprovação (ou não).

Análise 4: Relação Risco Região X Aprovação

Análise que consiste em verificar as relações entre o score de risco da região e o fato da aprovação(ou não).

Análise 5: Insuficiência de Saldo/Limite

Análise que consiste em verificar quantas transações são negadas por conta de saldo ou limite insuficiente.

Tabelas

Tabela 1: Transações

Tabela com N transações bancárias.
Forma de acesso: Arquivo CSV.

Tabela 1: Transações	
Descrição coluna	**Tipo**
ID da transação	string
ID do usuario pagador	string
ID do usuario recebedor	string
ID região	string
Modalidade do pagamento	string
Data e Horário	datetime
Valor da Transação	float

Tabela 2: Informações de cadastro

Tabela com as informações de cadastro de cada usuário.
Forma de acesso: SQLite

Tabela 2: Informações de cadastro	
Descrição coluna	**Tipo**
ID usuário	string
ID região	string

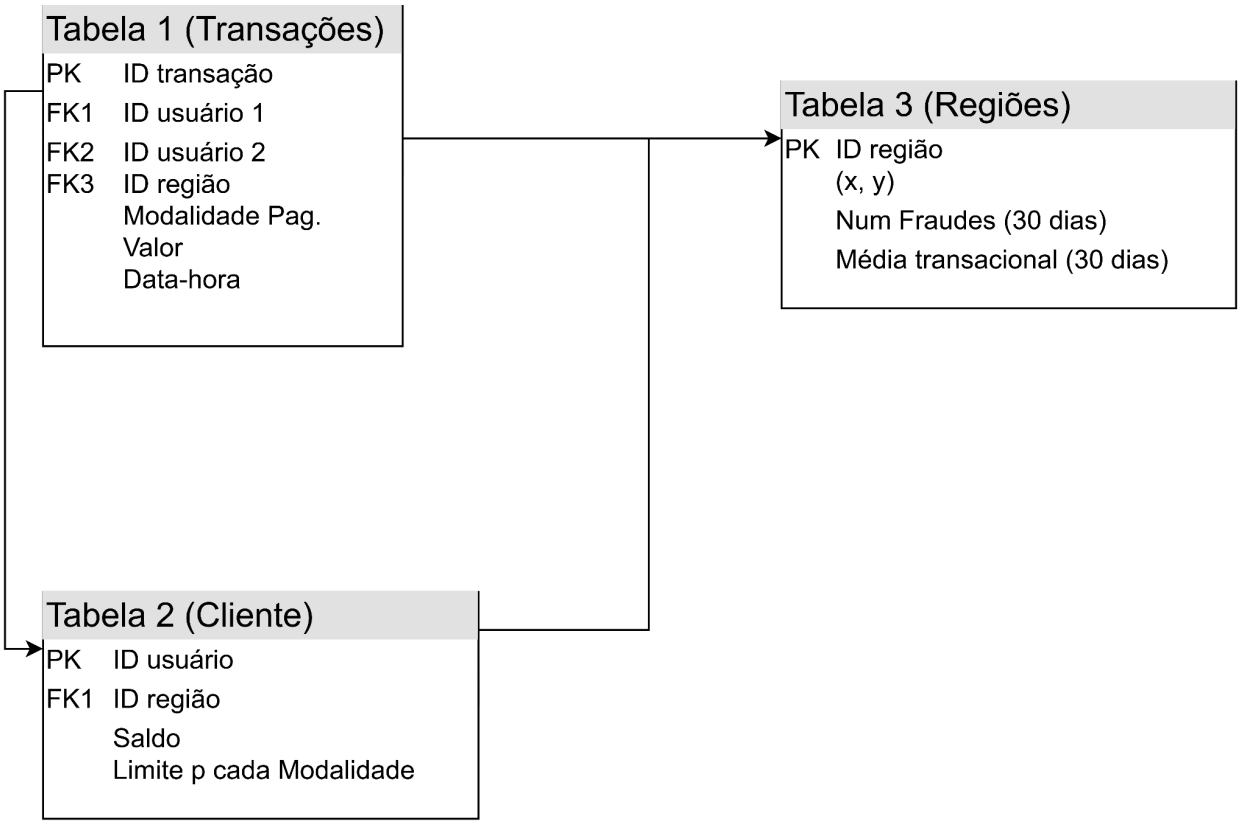
Tabela 2: Informações de cadastro	
Descrição coluna	**Tipo**
Saldo	float
Limite PIX	float
Limite TED	float
Limite CREDITO	float
Limite Boleto	float

Tabela 3: Regiões

Tabela com informações geográficas e transacionais de cada região.
Forma de acesso: CSV.

Tabela 3: Regiões	
Descrição coluna	**Tipo**
ID região	string
Latitude	float
Longitude	float
Média transacional (mensal)	float
Num fraudes (últimos 30 dias)	int

Esquema relacional



ETL

Extract

Temos três extratores, um para cada tabela do problema:

- E1 - Carrega os dados da Tabela 1
- E2 - Carrega os dados da Tabela 2
- E3 - Carrega os dados da Tabela 3

Transform

Como explicado na descrição do objetivo, para resolver o problema precisamos fazer verificações determinísticas e outras que se baseiam no “cálculo de risco”.

Operações determinísticas

- Verificar o saldo dos usuários
- Verificar limite de transação

Cálculo de risco:

- Risco pelo valor da transação

- Risco pelo horario
- Risco pela localização
- Função dos riscos acima

Tratador T1

Cria uma tabela a partir das associações dos IDs de usuário das tabelas vindas de E1 e E2 e adiciona a essa tabela uma coluna booleana “Aprovação” para definir, ao final da pipeline, quais transações serão aprovadas. Essa coluna é inicializada com todos os registros TRUE.

- Entrada: Tabelas vindas de E1 e E2.
- Saída: Tabela com colunas: ID transação, ID usuário, Modalidade pagamento, Valor transação, Saldo do usuário, Data-hora da transferência, Limites do usuário, ID Região Transação, ID Região Usuário, Aprovação, Limite Pix, Limite TED, Limite Crédito, Limite Boleto.

Tratador T2

Tratador responsável por verificar o saldo do usuário. Não utilizado no caso de transação com crédito, — essas são tratadas apenas no tratador T3 (responsável pelo limite) — por isso inicialmente filtra a tabela vinda de T1 para as transações diferentes de crédito. A partir da tabela de T1, verifica se o saldo do usuário é maior que o valor da transação requisitada. Em caso positivo, mantém-se TRUE. Em caso negativo, altera para FALSE o valor dos respectivos IDs na coluna “Aprovação”.

- Entrada: Tabela vinda de T1.
- Saída: Tabela no mesmo formato da recebida (Sem filtragem), porém com atualização nos valores da coluna “Aprovação” ,Limite Pix, Limite TED, Limite Crédito, Limite Boleto.

Tratador T3

Tratador responsável por verificar o limite de transações do usuário para a modalidade requisitada. Funciona de forma semelhante a T2, porém filtrando a tabela recebida para ter apenas as transações aprovadas. Em caso positivo, mantém-se TRUE . Em caso negativo, altera para FALSE o valor dos respectivos IDs na coluna “Aprovação”.

- Entrada: Tabela vinda de T2 (mesmo formato da tabela de T1).
- Saída: Tabela no mesmo formato da recebida (Sem filtragem), porém com atualização nos valores da coluna “Aprovação”,Limite Pix, Limite TED, Limite Crédito, Limite Boleto
- Análise: A tabela de saída será usada também para a análise (5).

Tratador T4

Cria uma tabela com a associação das tabelas vindas do extrator E3 e do tratador T1 de acordo com as variáveis de localização.

- Entrada: Tabelas vindas do Extrator E3 e do tratador T1.

- Saída: Tabela com colunas: ID transação, ID usuário, ID Região Transação, (x, y) Transação, ID Região Usuário, (x, y) Usuário.

Tratador T5

Recebe a tabela de T4 e calcula um score de risco para cada transação de acordo com as localizações.

- Entrada: Tabela vinda de T4.
- Saída: Uma tabela com os IDs de transação e o score de risco associado-o a cada ID.

Tratador T6

Recebe a tabela de T1 e calcula um score de risco para cada transação de acordo com o valor (se é muito alto ou muito mais alto que o comum do respectivo usuário).

- Entrada: Tabela vinda de T1.
- Saída: Uma tabela com os IDs de transação e o score de risco associado-o a cada ID.

Tratador T7

Recebe a tabela de T1 e calcula um score de risco para cada transação de acordo com o horário da transação.

- Entrada: Tabela vinda de T1.
- Saída: Uma tabela com os IDs de transação e o score de risco associado-o a cada ID.

Tratador T8

Soma os scores de T5, T6 e T7 e faz para cada ID: “Aprovação” = FALSE se score < τ; “Aprovação” = TRUE se score > τ (τ a definir).

- Entrada: 3 tabelas de scores vindas de T5, T6 e T7.
- Saída: Série com os IDs das transações e os dados da coluna “Aprovação”.
- Análise: Tem como saídas (além da principal) as tabelas para as análises (2), (3) e (4) que contém as colunas “score de risco”(de valor, de horário, de região, respectivamente) e “Aprovação”, Limite Pix, Limite TED, Limite Crédito, Limite Boleto.

Tratador T9

Junta os valores das colunas de aprovação vindos de T3 e T8 operando um end (FALSE + FALSE/TRUE = FALSE; TRUE + TRUE = TRUE).

- Entrada: Recebe a tabela de T3 (mesmo formato da de T1) e a série de T8
- Saída: Tabela no mesmo formato da recebida de T3, porém com atualização nos valores da coluna Aprovação, Limite Pix, Limite TED, Limite Crédito, Limite Boleto.

Tratador T10

Recebe a tabela de T9 e, — agrupando pelos IDs de usuários — onde se o somatório das transações for maior que o saldo, altera todos os valores referentes a esse ID na coluna aprovações para FALSE.

- Entrada: Tabela de dados de T9.
- Saída: Tabela no mesmo formato da entrada.

Tratador T11

Atualiza os saldos e os limites dos usuários a partir da tabela vinda de T10 e passa para os loaders duas tabelas.

- Entrada: Tabela de dados de T10.
- Saída: Tabela (1) com colunas “ID de transação” e “Aprovação” (semelhante à tabela saída de T8); Tabela (2) com colunas “ID de usuário”, “Saldo” e “Limite”.
- Análise: A tabela (1) é usada para analisar a presença de transações fraudulentas.

Load

Loader L1

Leva à uma base de dados do banco quais aprovações serão autorizadas.

- Entrada: Tabela (1) vinda de T11.
- Saída: A tabela de entrada carregada para a base de dados.

Loader L2

Recebe de T11 a tabela com usuários, saldos e limites e atualiza os respectivos dados na base de dados original (Tabela 2).

- Entrada: Tabela (2) vinda de T11.
- Saída: Atualização da tabela 2 da base de dados do banco.

Loader L3

Recebe de T8 a tabela com a coluna com o score de risco calculado para o valor e a de aprovação, e a carrega a uma base de dados para que seja feita sua respectiva análise.

- Entrada: Tabela com colunas (“score de risco (valor)”, “Aprovação”) vinda de T8.
- Saída: A tabela de entrada carregada para a base de dados.

Loader L4

Recebe de T8 a tabela com a coluna com o score de risco calculado para o horário e a de aprovação, e a carrega a uma base de dados para que seja feita sua respectiva análise.

- Entrada: Tabela com colunas (“score de risco (horário)”, “Aprovação”) vinda de T8.
- Saída: A tabela de entrada carregada para a base de dados.

Loader L5

Recebe de T8 a tabela com a coluna com o score de risco calculado para as regiões e a de aprovação, e a carrega a uma base de dados para que seja feita sua respectiva análise.

- Entrada: Tabela com colunas (“score de risco (região)”, “Aprovação”) vinda de T8.
- Saída: A tabela de entrada carregada para a base de dados.

Loader L6

Recebe a tabela saída de T3 e a carrega a uma base de dados para que seja feita sua respectiva análise.

- Entrada: Tabela vinda de T3.
- Saída: A tabela de entrada carregada para a base de dados.

Diagrama ETL

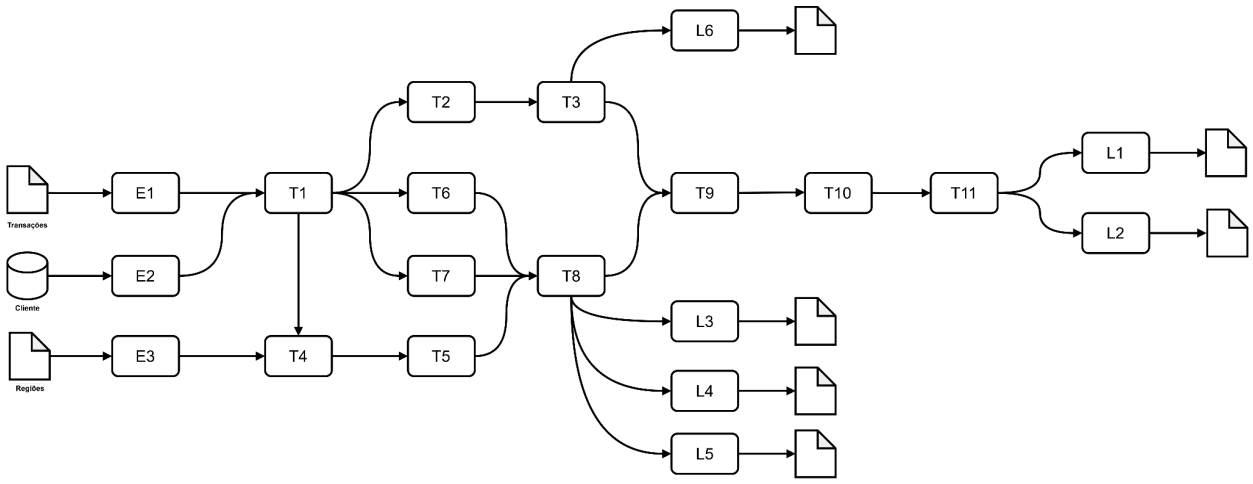
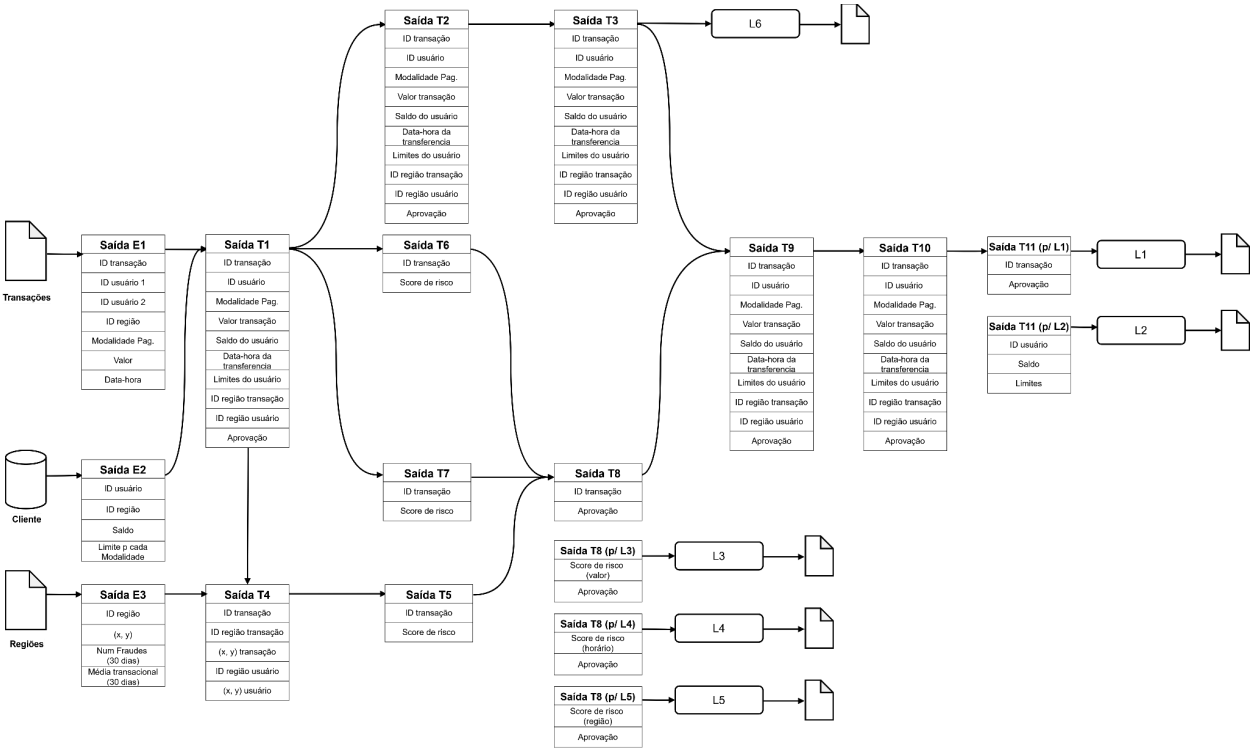


Diagrama ETL com as saídas (tabelas)

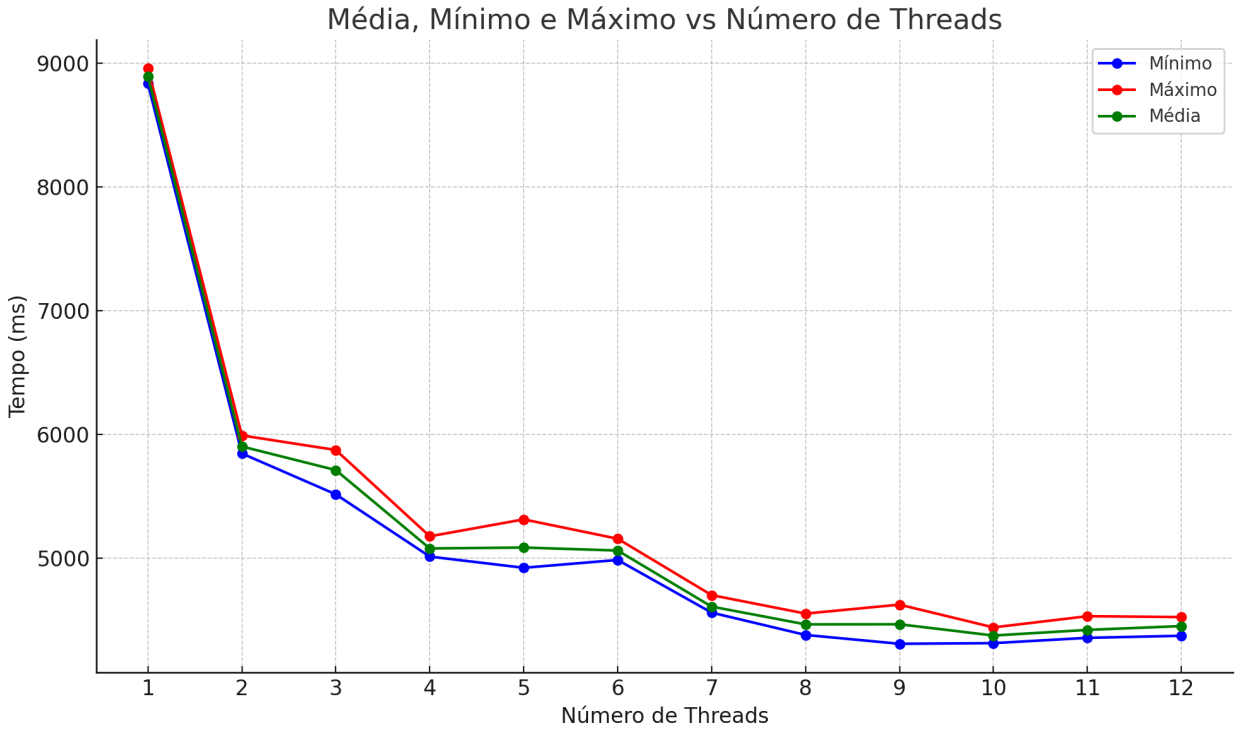

Transações
Saída E1
ID transação
ID usuário 1
ID usuário 2
ID região
Modalidade Pag.
Valor
Data-hora
Cliente
Saída E2
ID usuário
ID região
Saldo
Limite p cada Modalidade
Regiões
Saída E3
ID região
(x, y)
Num Fraudes (30 dias)
Média transacional (30 dias)
Saída T1
ID transação
ID usuário
Modalidade Pag.
Valor transação
Saldo do usuário
Data-hora da transferencia
Limites do usuário
ID região transação
ID região usuário
Aprovação
Saída T4
ID transação
ID região transação
(x, y) transação
ID região usuário
(x, y) usuário
Saída T2
ID transação
ID usuário
Modalidade Pag.
Valor transação
Saldo do usuário
Data-hora da transferencia
Limites do usuário
ID região transação
ID região usuário
Aprovação
Saída T6
ID transação
Score de risco
Saída T7
ID transação
Score de risco
Saída T5
ID transação
Score de risco
Saída T3
ID transação
ID usuário
Modalidade Pag.
Valor transação
Saldo do usuário
Data-hora da transferencia
Limites do usuário
ID região transação
ID região usuário
Aprovação
Saída T8
ID transação
Aprovação
Saída T8 (p/ L3)
Score de risco (valor)
Aprovação
Saída T8 (p/ L4)
Score de risco (horário)
Aprovação
Saída T8 (p/ L5)
Score de risco (região)
Aprovação
L6
L3
L4
L5
Saída T9
ID transação
ID usuário
Modalidade Pag.
Valor transação
Saldo do usuário
Data-hora da transferencia
Limites do usuário
ID região transação
ID região usuário
Aprovação
Saída T10
ID transação
ID usuário
Modalidade Pag.
Valor transação
Saldo do usuário
Data-hora da transferencia
Limites do usuário
ID região transação
ID região usuário
Aprovação
Saída T11 (p/ L1)
ID transação
Aprovação
Saída T11 (p/ L2)
ID usuário
Saldo
Limites
L1
L2

Resultados

Processamento de tabelas de usuários e transações com 100 mil linhas cada.

# Threads	Exec1	Exec2	Exec3	Exec4	Exec5	Média	Mínimo	Máximo
1	8957.48	8879.98	8834.15	8912.36	8874.02	8891.60	8834.15	8957.48
2	5993.02	5913.39	5895.28	5847.33	5870.04	5903.812	5847.33	5993.02
3	5811.92	5526.59	5517.19	5875.27	5834.05	5713.004	5517.19	5875.27
4	5096.02	5178.16	5045.54	5014.44	5066.78	5080.188	5014.44	5178.16
5	5119.11	5067.68	4923.99	5011.88	5314.94	5087.520	4923.99	5314.94
6	5158.78	5074.33	5005.62	5089.01	4987.21	5063.390	4987.21	5158.78
7	4703.33	4561.87	4589.46	4592.95	4603.30	4610.582	4561.87	4703.33
8	4554.12	4473.10	4482.45	4446.38	4381.53	4467.116	4381.53	4554.12
9	4310.21	4412.41	4465.56	4626.35	4524.41	4467.788	4310.21	4626.35
10	4442.59	4397.40	4409.34	4315.32	4319.85	4376.900	4315.32	4442.59
11	4358.58	4358.20	4481.86	4532.96	4379.70	4422.260	4358.20	4532.96
12	4525.76	4480.64	4471.38	4412.66	4374.88	4453.064	4374.88	4525.76

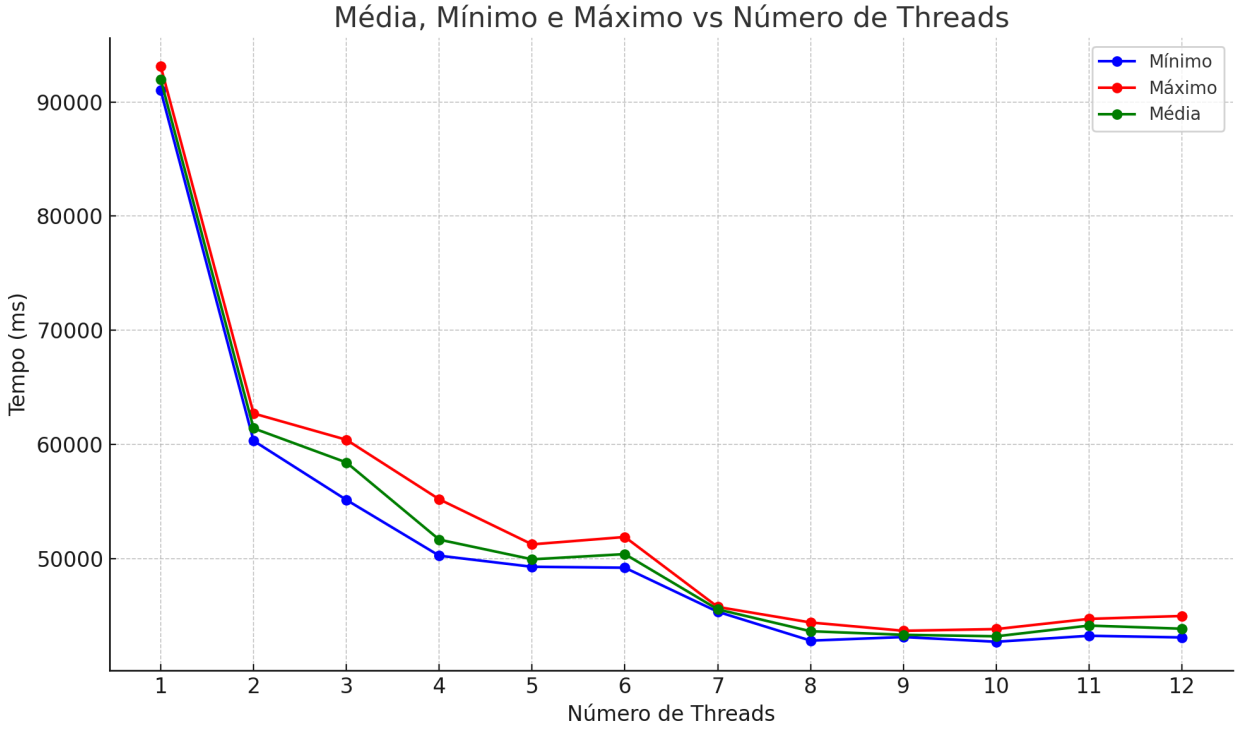
Table 1: Tempos de execução (ms) por número de threads



Processamento de tabelas de usuários e transações com 1 milhão de linhas cada.

# Threads	Exec1	Exec2	Exec3	Exec4	Exec5	Média	Mínimo	Máximo
1	93114.50	92692.70	91002.40	91522.40	91548.40	91976.08	91002.40	93114.50
2	60984.90	60336.00	60628.70	62728.10	62461.00	61427.74	60336.00	62728.10
3	57685.10	59116.10	60425.00	59806.40	55160.70	58438.66	55160.70	60425.00
4	55214.00	50977.40	50913.60	51029.10	50275.80	51681.98	50275.80	55214.00
5	49305.70	49467.80	49370.90	51268.80	50378.40	49958.32	49305.70	51268.80
6	51913.80	51151.80	49663.70	50112.10	49222.40	50412.76	49222.40	51913.80
7	45364.40	45686.00	45368.60	45672.70	45788.50	45576.04	45364.40	45788.50
8	43926.10	42840.50	44428.30	43384.10	43725.80	43660.96	42840.50	44428.30
9	43154.30	43427.90	43701.60	43262.90	43198.90	43349.12	43154.30	43701.60
10	42738.60	43167.30	43581.10	42754.50	43850.90	43218.48	42738.60	43850.90
11	44745.30	44001.90	43264.20	44119.10	44650.80	44156.26	43264.20	44745.30
12	44998.00	44792.10	43300.10	43119.10	43206.70	43883.20	43119.10	44998.00

Table 2: Tempos de execução (ms) por número de threads



Conclusão

As tabelas mostram os tempos de execução da pipeline (com dados gerados pelo mock) com diferentes quantidades de threads. À luz disso, perceba que, ao passo que variamos as threads usadas no processamento, observa-se uma redução significativa no tempo médio de execução,

especialmente entre 1 e 4 threads. Por outro lado, a partir de 8 threads, o acréscimo de mais threads não resulta em melhorias. Concluímos que nossa implementação foi capaz de explorar de forma eficiente os recursos computacionais disponíveis na máquina, utilizando corretamente o paradigma de paralelismo com múltiplas threads.