Pipeline ETL para processamento de transacional bancário via RPC

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

1. Introdução

O presente relatório é referente à atividade A2 da 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 pela descrição de nossas alterações e modelagem para a atividade extra.

Relembrando a entrega anterior (A1):

- Todo o ETL feito em C++;
- Temática: Transações bancárias.

Abordagem da entrega RPC: Cliente em Python e Servidor em C++.

Fluxo de Dados do Cliente ao Pipeline ETL:

1. **Envio Contínuo do Cliente:** Cada cliente envia periodicamente, via streaming, dados estruturados conforme definido no Proto (futuras linhas do dataframe). A interface RPC oferece apenas um método, `SendTransactions,` que recebe o stream de transações, e ao final do stream retorna um OK.
2. **Recepção e Acúmulo no Servidor:** Uma thread dedicada no servidor recebe o fluxo de dados de cada cliente e armazena as linhas em um vetor específico para aquele cliente.
3. **Criação de Batches:** O servidor forma um batch de dados para um cliente quando atinge um limite de X linhas recebidas OU quando o intervalo entre envios excede um tempo Y. Esse batch é então adicionado à fila do gerenciador da pipeline através de um método da classe `PipelineManager`.
4. **Gerenciamento da Pipeline:** O `PipelineManager` (executado por uma única thread) consome os batches da fila e os adiciona a um dataframe.
5. **Processamento do Dataframe:** O `PipelineManager` processa o dataframe utilizando o `ServerTrigger` (uma adaptação do `RequestTrigger` da entrega A1). O processamento ocorre sempre que a Pipeline ETL não está em execução e o tamanho do dataframe ultrapassa um limite mínimo predefinido.

2. Arquitetura do Cliente Python

O cliente Python, encapsulado na classe `TransactionClient`, é projetado para simular a geração e o envio de transações bancárias para um servidor gRPC. Sua principal função é testar a capacidade do servidor de receber e processar esses dados, com foco atual no envio via streaming.

Propósito Principal

- Gerar dados de transações bancárias simuladas com características realistas (IDs, valores, datas, métodos de pagamento, regiões).
- Enviar essas transações para um servidor gRPC para processamento.
- Oferecer flexibilidade na configuração (endereço do servidor, número de transações, semente para aleatoriedade) através de argumentos de linha de comando.

Geração de Transações (`generate_transaction`)

- Cria dinamicamente dados para cada transação.
- Os dados são formatados em uma mensagem Protobuf.
- Um timestamp é adicionado à mensagem no momento da sua criação.

Comunicação gRPC

- Estabelece um canal de comunicação com o endereço do servidor especificado.
- Cria um `stub` para interagir com o serviço `TransactionService` definido no servidor.

Modo de Envio Principal (Streaming de Cliente)

- **Iterador de Transações (`transaction_iterator`):** Implementado como um iterador Python (com `yield`), produz sequencialmente as transações, sob demanda.
- **Envio em Stream (`sender_thread`):**
    - Invoca o método RPC `self.stub.SendTransaction()` passando o `transaction_iterator` como argumento.
    - Isso indica que o método `SendTransaction` no servidor é projetado para receber um fluxo (stream) de mensagens `Transaction` do cliente. O cliente envia as transações uma a uma conforme são geradas pelo iterador, sem precisar carregar todas em memória de uma vez.
- A função `main()` atualmente prioriza este modo de operação, chamando `client.sender_thread()`.

5. Modo de Envio Alternativo/Legado (não compatível com o proto final):

- **Envio Individual (`send_transaction`):** Envia uma única transação para o servidor (presumivelmente usando uma chamada RPC unária) e aguarda uma resposta, medindo a latência.
- **Execução Paralela (`run`):**

- Utiliza `concurrent.futures.ThreadPoolExecutor` para enviar múltiplas transações individuais em paralelo.
- Cada transação gerada é submetida como uma tarefa separada para o pool de threads, que chama `send_transaction`.
- Este modo permite testar o envio de múltiplas requisições unárias concorrentes.

Em resumo, o `TransactionClient` é uma ferramenta de simulação que estabelece comunicação com um servidor gRPC e envia dados de transações, primariamente através de um mecanismo de streaming de cliente, onde múltiplas transações são enviadas em um único fluxo contínuo. Ele também retém a capacidade de enviar transações individualmente de forma paralela para outros cenários de teste.

Inicialmente havíamos optado pelo método proto ser chamado *a cada transação enviada ao servidor*, mas após ponderarmos sobre o tempo associado à criação e remoção de conexões, concluímos que seria mais interessante ter o método recebendo uma *stream* de transações.

3. Arquitetura do Servidor C++

O servidor C++ foi projetado para receber fluxos (streams) de transações bancárias de múltiplos clientes via gRPC. Ele processa essas transações em lotes (batches) e dispara uma pipeline de processamento para cada lote que atinge um tamanho pré-definido.

O código associado ao servidor tem como principal propósito atuar como um endpoint gRPC para clientes Python, decodificando a stream de transações em um formato mais amigável e enviando a carga recebida para um objeto dedicado que gerencia a pipeline.

A classe `TransactionServerImpl`, que é base para o servidor, herda de `transaction::TransactionService::Service`, que por sua vez é uma classe dos stubs gerados pelo gRPC seguindo a interface definida por nós no arquivo `.proto`. Essa classe utiliza do programa gerado pelo gRPC para fazer o gerenciamento das conexões do servidor. O que é implementado por nós é o método do RPC.

O método chave é `Status SendTransaction()`, definido na classe `TransactionServerImpl.` Este método é invocado pelo framework gRPC quando um cliente inicia um envio de stream de transações, e é efetivamente a implementação do método RPC do arquivo proto. Como os inputs se tratam de stream de dados contínuos, a biblioteca gRPC inicia uma thread nova para cada conexão, para ficar tratando das transações recebidas. Devido à natureza da implementação da biblioteca, não é possível, usando a interface padrão, controlar quantas threads são iniciadas pelo gRPC.

Esse método utiliza a interface de stream da biblioteca para ler sequencialmente as mensagens `Transaction` enviadas pelo cliente. Um loop processa cada transação

recebida na stream, e elas são todas convertidas para vetores de `std::variant` (que definimos como o tipo `VarRow` por conveniência) e adicionados a um vetor.

Quando o vetor de `VarRow` atinge um tamanho específico OU quando o intervalo entre envios de transações pela stream excede um tempo pré-determinado, sua carga é transferida para o objeto `PipelineManager`, que internamente mantém uma fila de vetores de `VarRow`, como será melhor explicado na seção dedicada a esse objeto. Após transferir a carga, o vetor é esvaziado e a thread segue salvando as transações recebidas pelo cliente.

Caso a stream seja finalizada (ou seja, o cliente foi desligado), o servidor envia uma resposta simples de OK e a thread associada àquele cliente é encerrada automaticamente pela saída de escopo.

4. Gerenciador de Pipeline

O `PipelineManager` é uma classe projetada para gerenciar de forma desacoplada o fluxo de dados entre produtores de dados (nesse caso, um servidor gRPC) e a pipeline de processamento de dados. Sua principal responsabilidade é receber lotes de dados, no formato de vetores de transações (como dito anteriormente, representadas pelo tipo definido `VarRow`), agregá-los em no formato de dataframe, que é processado pela pipeline. e disparar a execução da pipeline com os dados agregados quando ela terminar um processamento anterior.

Para ser capaz de seguir essas funções, essa classe é associada à outras 3:

- Um vetor de vetores de `VarRow`. O servidor RPC escreve, como citado anteriormente, lotes de transações nesse vetor, que é lido regularmente pelo gerenciador da pipeline, que vai serializando as transações dentro de um dataframe.
- Dois dataframes, que têm a mesma estrutura colunar da tabela de transações da pipeline. Um deles é o dataframe que é lido pela pipeline, e o outro é populado enquanto a pipeline está em execução.
- Um `ServerTrigger`, tipo especial de trigger adicionado para esta entrega que serve de interface para a pipeline de processamento, sendo o recurso utilizado pelo gerenciador para iniciar a pipeline.

A inserção de lotes de transações é feita por meio de um método thread-safe (protegido por um mutex), `submitDataBatch`, permitindo que as threads do servidor gRPC acumulem lotes de dados na classe. Esses lotes são adicionados em um vetor, como explicado acima.

Dentro do loop interno do gerenciador, que é executado em uma thread dedicada, o seguinte processo ocorre:

- O loop acessa o primeiro lote da fila interna e serializa as transações para o dataframe que está sendo populado. Esse dataframe não é lido pela pipeline, o que

garante que a thread do gerenciador pode escrever nele enquanto a pipeline é executada.
- Após serializar um lote, a thread verifica se a pipeline está funcionando, por meio de um atributo do `ServerTrigger` (será explicado mais à frente).
    - Se a pipeline estiver sendo executada, o loop volta para o primeiro passo e serializa mais um lote de transações para o dataframe que está sendo populado.
    - Se a pipeline não estiver sendo executada, o dataframe é passado para ela e ela é iniciada em uma thread separada, onde irá processar os dados. Após isso, o dataframe interno que é populado é trocado por um que não será lido pela pipeline.
- Independente de ter executado ou não a pipeline, o loop segue adicionando as transações em um dataframe que não é lido pela pipeline no momento. Existe um mecanismo com uma `condition_variable` para evitar que a thread não fique em espera ociosa quando não há lotes para serializar.

5. Adaptações em triggers e extratores

Para que esse sistema de gerenciamento funcionasse, foi necessário introduzir alterações nos triggers e extratores para:

- Permitir que dados fossem lidos de dataframe já inicializados, sem a necessidade de passar por outros meios de leitura;
- Permitir que extratores que sempre liam a mesma base pulassem a etapa de leitura em execuções consecutivas da pipeline;
- Permitir que uma thread sendo executada paralelamente à pipeline pudesse averiguar seu funcionamento, por meio do trigger.

As alterações feitas se concentraram em duas classes: a nova classe `ServerTrigger` e a nova classe `ExtractorNoop`.

A classe `ExtractorNoop` não é associada à repositórios de dados externos, e sim diretamente a um dataframe. Essa natureza foi necessária para poder executar a pipeline com os dados de transação acumulados em dataframe sem a necessidade de escrever em disco essas transações desnecessariamente.

Além dessa nova classe, a classe `Extractor` foi acrescida de uma nova flag que sinaliza que ela não deve apagar seus dados entre execuções da pipeline, evitando que a leitura em disco da mesma informação seja feita todas as vezes.

A classe `ServerTrigger` foi introduzida para tornar a interface com o gerenciador da pipeline mais limpa. Ela possui duas diferenças em relação ao `RequestTrigger`:

- Ela fornece um parâmetro para outra thread verificar se a pipeline está em execução;
- Seu método `start`, além do número de threads, recebe um dataframe, e antes de iniciar a execução, ela faz a ligação de um `ExtractorNoop` com o dataframe, garantindo a exeução com os dados corretos.

6. Vantagens e desvantagens

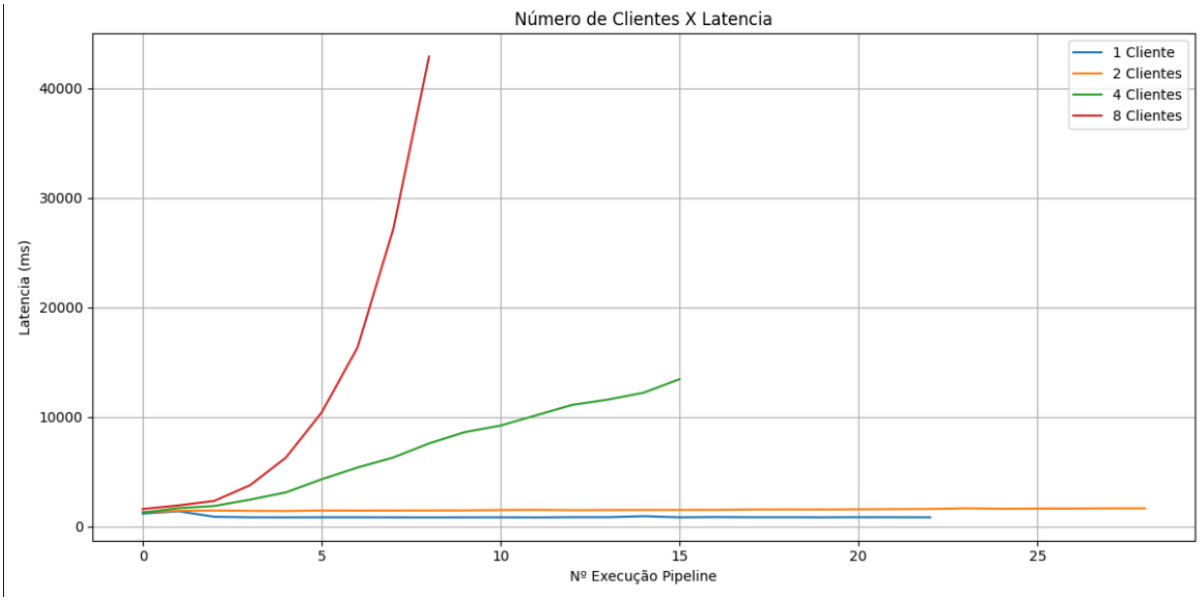
Para nossa entrega, escolhemos a abordagem que faz o servidor RPC em C++. As vantagens dessa abordagem incluem:

- Concentração do servidor e da pipeline em um único processo, facilitando o gerenciamento de threads e o acesso de memória compartilhada;
- Maior agilidade ao tratar os dados, já que não é necessário escrevê-los em disco ou serializa-los novamente em um formato de mensagem agnóstico à linguagem;
- Melhor integração do desenvolvimento, já que todos os componentes são escritos na mesma linguagem e podem compartilhar interfaces de maneira mais fácil. Executar a pipeline de um script python não permitiria uma boa integração das bases de código.
- Maior controle dos recursos do sistema, já que C++ por natureza é uma linguagem de mais baixo nível do que Python. Isso permite a escrita de rotinas mais otimizadas e rápidas.

Já as principais desvantagens da abordagem são:

- Necessidade de armazenar todos os dados recebidos em memória, possibilitando a perda de dados caso o processo seja interrompido abruptamente e houvesse uma fila de execução;
- Risco da fila de dados a serem tratados exceder a memória disponível no sistema, o que pode levar a erros de alocação e consequentemente impossibilitar a leitura de mais dados recebidos pelas streams;
- Maior risco a falhas de desenvolvimento, por C++ se tratar de uma linguagem mais insegura que em comparação à Python. Erros no desenvolvimento podem ser encontrados apenas em casos especiais não cobertos pelos testes feitos no desenvolvimento.

7. Resultados



Este gráfico mostra a latência média (latência = momento em que a pipeline terminou de processar a transação - momento em que ela foi iniciada, em milissegundos) dos registros que eram processados por chamada da pipeline, para diferentes números de clientes gerando registros.
Como é possível observar pelo plot, nossa implementação consegue manter uma latência constante entre emissão da transação e sua validação pela pipeline quando estamos trabalhando com 1, e 2 clientes conectados ao servidor (mas evidentemente essa latência estável aumenta de acordo com o número de clientes). Essa estabilidade não foi alcançada com 4 e 8 clientes, já que a latência começou a crescer consecutivamente entre cada execução da pipeline.
Possíveis direções para melhorar isso incluiriam:

- Melhor escolha dos valores mínimos de transações e de tempos para as threads inserirem lotes de linhas na fila do gerenciador;
- Melhor escolha dos valores usados pelo orquestrador da pipeline para pré-alocar o número de threads por tratador.

Dar mais atenção para esses dois pontos permitiria um melhor desempenho da pipeline para a carga dos clientes.