Fundação Getúlio Vargas
Escola de Matemática Aplicada

Ciência de Dados e Inteligência Artificial

Computação Escalável

Alessandra Belló Soares
Kaiky Eduardo Alves Braga
Larissa Lemos Afonso
Luciano Pereira Sampaio
Samuel Corrêa Lima

Rio de Janeiro
Abril / 2025

Sumário

1 Estrutura do DataFrame

Para a construção inicial do DataFrame, foi utilizada a classe de tipagem Variant, que permite a atribuição de diferentes tipos de valores a uma mesma variável. Com isso, os dados no DataFrame são armazenados como vetores de vetores por coluna, em que cada elemento pode assumir tipos definidos pela classe Variant.

Além dos dados, o DataFrame também mantém metadados importantes, como o número de colunas e registros, os nomes das colunas e os tipos associados a cada uma delas.

Além disso, a implementação inclui mecanismos de concorrência: um mutex global para o DataFrame e mutexes específicos para seus registros e colunas.

Nele, estão presentes os seguintes métodos:

- addColumn : Adiciona uma coluna ao DataFrame.
- addRecord : Adiciona um registro no DataFrame.
- addMultipleRecords : Adiciona um bloco de registros no DataFrame.
- getRecords : Retorna um novo DataFrame de acordo com o vetor de índices de registros passados.
- printDF : Realiza o print tabular do DataFrame.
- DFtoCSV : Realiza a conversão para CSV.
- getRecord : Retorna o registro de uma linha.
- getNumCols : Retorna o número de colunas.
- getColumn : Retorna a coluna dado seu nome.
- getColumnTypes : Retorna o tipo das colunas.
- getRecords : Retorna o número de registros.
- getColumnIndex : Retorna o índice refrente à uma coluna.
- changeColumnName : Altera o nome de uma coluna.

Para inicialização do DataFrame, basta passar como entrada dois vetores de string, um contendo o nome das colunas e outro contendo o tipo delas (int, float, string, bool).

Obs: Vale ressaltar que os mutex são utilizados frequentemente nesses métodos para evitar conflitos entre duas tarefas envolvendo o mesmo DataFrame.

2 Extratores

2.1 SQLite

O extrator de dados tem como objetivo ler todas as informações de uma tabela do SQLite e preencher o DataFrame em memória de forma eficiente. O funcionamento do extrator pode ser dividido em duas partes principais que ocorrem simultaneamente, a de leitura e a de processamento.

Na parte da leitura, somente uma thread é responsável por acessar o banco de dados e ler os registros sequencialmente. Essa leitura chama a função de callback a cada linha lida. As linhas lidas são armazenadas temporariamente em um vetor chamado blockRead. Quando esse vetor atinge um tamanho mínimo definido, seu conteúdo é transferido para um vetor principal (linesRead). Esse processo de transferência é protegido por um mutex para garantir que a operação de adicionar novos dados ao vetor compartilhado ocorra de maneira segura.

Enquanto a leitura continua, cada thread monitora o vetor linesRead e, ao detectar que novos dados estão disponíveis, pegam um conjunto de linhas para processar, utilizando um contador de registros (recordsCount), também protegido por um mutex, para garantir que cada linha seja processada apenas uma vez. As threads copiam suas respectivas porções de dados para a memória local, liberam o acesso ao vetor compartilhado e então, trabalham na adição dos dados no DataFrame de forma independente.

O extrator continua operando até que todos os dados do banco tenham sido lidos e todas as linhas armazenadas tenham sido processadas. Quando essas duas condições são satisfeitas, a execução de todas as threads é encerrada, e o DataFrame finalizado fica disponível para ser utilizado em etapas posteriores da aplicação.

Obs: Como os blocos podem ser pegos em tempos diferentes pelas threads, a ordem pode não ser a mesma do banco de dados.

2.2 CSV

A lógica geral do extrator do CSV é parecida a do SQLite, onde uma thread é responsável por ler os dados (arquivo ou banco) e várias threads paralelas processam os registros conforme eles vão sendo lidos. Em ambos os casos, as linhas lidas são armazenadas em um vetor compartilhado, e o acesso concorrente a esse vetor é controlado por mutexes para evitar problemas de concorrência.

A principal diferença está na fonte dos dados e no método de processamento. No banco de dados, as linhas são extraídas diretamente da consulta SQL e armazenadas em blocos. Em ambos o processamento pode ser linha à linha ou em blocos (no geral o processamento em bloco é mais eficiente).

Diferente do leitor do SQLite, o leitor do CSV não passa os dados do vetor local blockRead para o vetor compartilhado linesRead simplesmente ao chegar em um valor mínimo de linhas, pois também considera o progresso dos processadores de linhas. Através de uma flag, os processadores conseguem avisar quando chegaram ao final do vetor linesRead e, somente quando avisado, o leitor passa o seu progresso em

blockRead para o vetor compartilhado linesRead. Dessa forma, o leitor não precisa esperar pela liberação do mutex com tanta frequência e, portanto, termina a leitura mais rapidamente.

Além disso, no extrator de banco o controle de fluxo da leitura é feito pelo próprio mecanismo do SQLite, enquanto no CSV controla manualmente a leitura até o fim do arquivo. Apesar das diferenças, o fluxo geral de leitura e paralelismo no processamento permanece o mesmo nos dois casos.

Obs: A flag de controle do leitor, quando implementada da mesma forma no extrator de SQL, reduziu o seu desempenho, e não conseguimos descobrir o motivo. Portanto, esse controle só está presente no leitor de CSV.

3 Triggers

Para o tratamento dos triggers, foi utilizado um ThreadPool, que é uma estrutura responsável por gerenciar um conjunto fixo de threads. Em vez de criar e destruir uma thread toda vez que surge uma tarefa nova, o pool mantém um grupo de threads reutilizáveis.

Quando uma nova tarefa precisa ser executada, ela é adicionada no pool usando o enqueue. Junto com a tarefa, também é informado um ID que representa quando ela poderá ser liberada para execução. Inicialmente, essas tarefas não são executadas em seguida elas ficam armazenadas em uma fila de waitingTasks, aguardando a liberação.

A liberação das tasks é feita pelo método isReady, que recebe um ID como argumento. Quando chamado, o isReady procura todas as tarefas com o ID correspondente e move elas para a fila principal de execução. Assim, só depois do isReady é que a task poderá realmente ser executada por uma das threads do pool. Como os IDs podem repetir, ao acionar isReady(id), a função aciona todas tarefas associadas ao identificador.

As threads ficam em um loop contínuo, esperando que novas tarefas apareçam na fila. Quando há uma tarefa pronta, uma thread a pega e executa a função associada. Todo o controle de acesso às filas é feito usando mutexes, para garantir que não haja problemas de concorrência, e o mecanismo de acordar as threads é feito com variáveis de condição.

Essa separação entre enfileirar a tarefa (enqueue) e liberar sua execução (isReady) ajuda no sistema de dependências entre as tarefas. Ou seja, uma tarefa só é liberada para execução depois que todas as tarefas das quais ela depende já tiverem sido concluídas. Garantindo assim, que a ordem de execução respeite as dependências necessárias.

O sistema também permite que o usuário receba o resultado da execução de uma task usando um future, que facilita sincronizar processos que precisam esperar pelo fim de uma tarefa.

Por fim, quando o ThreadPool é destruído, ele cuida para que todas as threads finalizem corretamente. Primeiro, ele sinaliza que o pool está parando, depois acorda todas as threads, e finalmente espera cada uma terminar sua execução antes de destruir os recursos. Esse processo evita que fiquem threads ativas ou que o programa

tenha vazamento de memória.

4 Tratadores

Todos os tratadores apresentados seguem um padrão de execução semelhante: cada função recebe o id do tratador, o número de threads usadas no tratador, um pool originado do sistema de thread pool (que será detalhado posteriormente) e divide o trabalho entre as threads, processando o DataFrame em blocos de tamanho similar. Cada thread realiza seu processamento localmente e, ao final, os resultados são combinados para formar a saída final. Além disso, todas as tarefas associadas aos blocos recebem um id (o negativo do id do tratador) para manusear no Thread Pool. A seguir, descrevemos o funcionamento específico de cada tratador:

4.1 Filter Records

A função filtra registros de um DataFrame com base em uma condição aplicada a suas colunas.

Inicialmente, utilizamos a função auxiliar filter block records, que recebe um DataFrame, uma condição e um intervalo de índices (mínimo e máximo), retornando os índices que satisfazem a condição dentro desse intervalo.

O DataFrame é particionado em blocos, cada thread processa seu bloco avaliando quais registros atendem à condição. Após a execução paralela, os vetores de índices filtrados são reunidos e ordenados.

Por fim, a função filter records by idxes é aplicada para criar um novo DataFrame contendo apenas os registros filtrados.

4.2 Group By Mean

A função realiza o agrupamento dos registros por uma coluna e calcula a média de outra coluna para cada grupo.

Primeiramente, extraímos os vetores de dados das colunas de interesse (coluna de agrupamento e coluna alvo). Em seguida, o DataFrame é particionado em blocos para as threads, e cada thread acumula a soma dos valores e a contagem de ocorrências para cada chave do agrupamento.

Após o processamento paralelo, os hash maps parciais são unidos, somando as somas e contagens correspondentes de cada chave.

Por fim, um novo DataFrame é construído, contendo cada chave de agrupamento e a média calculada (soma total dividida pela contagem total).

4.3 Join by Key

A função une dois dataframes baseado em uma coluna chave comum aos dois dataframes. Para executar essa união também dividimos o dataframe em blocos e cada thread executa um bloco.

O resultado final é um dataframe unico com todas as colunas dos dois dataframes iniciais.

4.4 Get Hour by Time

A função extrai as horas de colunas de horário formatadas como `HH:MM:SS`.

Cada thread percorre seu bloco de registros, extraindo os dois primeiros caracteres da string de horário (correspondentes à hora) e armazenando em um vetor local.

Após o processamento paralelo, os vetores de horas são combinados e é criada uma coluna contendo as horas extraídas, por fim ela é adicionada a um novo DataFrame que possui apenas ela.

4.5 Count Values

A função faz a contagem de valores de uma determinada coluna. Cada thread percorre seu bloco de registros e faz a contagem para cada valor do bloco em um dicionário, onde a chave é o elemento da coluna e o valor é a contagem dele.

Após o processamento paralelo, juntamos todos esses resultados em um novo dicionário e, a partir dele, criamos um novo DataFrame com esses resultados.

OBS: O Count Values possui um prâmetro opcional (numDays), que serve para calcular a média das contagens.

4.6 Sort by Column

A função implementa uma versão paralelizada do Merge Sort para ordenar um DataFrame com base em uma coluna especificada. O processo começa dividindo o DataFrame em blocos menores, que são atribuídos às diferentes threads disponíveis. Cada thread realiza a ordenação dos registros dentro do seu respectivo bloco de forma independente.

Após a ordenação local de cada bloco, a função utiliza o algoritmo de merge para combinar os blocos ordenados. O merge é feito com o auxílio de um heap, que permite comparar e juntar os elementos de maneira rápida, mantendo a ordenação correta.

4.7 Calculate Mean

A função realiza a média de uma determinada coluna.

Cada thread percorre seu bloco de registros na coluna, extraindo a soma parcial desse intervalo.

Após o processamento paralelo, os resultados parciais são somados e divididos pelo metadado do DataFrame do número de registros presentes.

4.8 Get Quantile

A função calcula os quantis de uma coluna numérica de um DataFrame. Primeiro, o DataFrame é ordenado e o número de registros também é extraído.

Dentro da função, é criada uma função auxiliar que calcula o valor de cada quantil a partir da coluna ordenada. Primeiro, a posição do quantil é determinada multiplicando o valor do quantil pelo número de registros menos um. Se a posição for um número inteiro, o valor do índice correspondente é retornado diretamente, caso contrário, é feita uma interpolação linear entre os valores dos índices inferior e superior. Se os valores interpolados forem inteiros ou floats, o resultado respeita esse tipo, para outros tipos, simplesmente é retornado o valor inferior.

Depois de definir essa função auxiliar, a função principal percorre a lista de quantis solicitados. Para cada quantil, cria um rótulo específico ("min"para 0.0, "median"para 0.5, "max"para 1.0 e "QXX"para os demais) e associa o rótulo ao valor calculado. Esses pares são armazenados em um hash map, que é retornado no final.

5 Resultados

5.1 Classify Accounts

Essa função realiza a classificação de contas em categorias "A", "B", "C"ou "D"com base em duas métricas: a média de transações (class first) e o saldo (class sec).

Primeiramente, as colunas "id col", "media col"e "saldo col", são extraídas do DataFrame.

Em seguida, cada thread processa seu bloco:

- Valida os tipos dos elementos de cada linha.
- Classifica cada conta:
  - "A"se a média for superior a 500.
  - "B"se a média estiver entre 200 e 500 e o saldo for maior que 10.000.
  - "C"se a média estiver entre 200 e 500 e o saldo for menor ou igual a 10.000.
  - "D"se a média for inferior a 200 .
- Armazena os IDs e as categorias localmente.

Por fim, os resultados são unidos e geram um DataFrame contendo as colunas"account id"e "categoria".

5.2 Top 10 cidades

A função retorna um DataFrame contendo as top 10 cidades com mais transações.

Primeiro se usa o count values para contar quantas transações ocorreram em cada cidade. Em seguida, o DataFrame resultado é ordenado de maneira decrescente.

Por fim, um novo DataFrame contendo as top 10 cidades é criado com as colunas "location"e "num trans".

5.3 Abnormal Transactions

A função serve para encontrar transações que parecem estranhas. Ela olha para dois tipos de problemas, transações com valores muito fora do normal e transações feitas em cidades diferentes da cidade da conta.

Para o valor, a função calcula um intervalo que representa o que seria considerado normal (usando o intervalo IQR). Se uma transação for muito maior ou muito menor do que esse intervalo, ela é marcada como anormal.

Para a cidade, a função compara a cidade onde a transação aconteceu com a cidade cadastrada da conta. Se forem diferentes, a transação também é considerada suspeita (Já que em 98% dos casos as pessoas fazem transações na cidade de origem).

Assim, a função divide o trabalho em threads para checar vários dados ao mesmo tempo. No final, todas as transações suspeitas são juntadas em um único resultado.

5.4 Summary Stats

A função calcula estatísticas de uma coluna específica de um DataFrame. Primeiro, ela chama o tratador getQuantiles para calcular os quantis solicitados da coluna. Após isso, a função converte os valores inteiros de quantis para o tipo float. Em seguida, a média dos valores na coluna é calculada utilizando o tratador calculateMeanParallel.

Por fim, a função retorna um DataFrame com as estatísticas calculadas, organizado com as colunas "statistic"e "value", onde cada linha contém o nome da estatística e seu respectivo valor.

5.5 Número de Transações por hora

A função precisa utilizar dois tratadores: Get Hour By Time e Count Values. É utilizado Get Hour By Time para pegar apenas as horas da coluna de tempo e depois é utilizado Count Values para obter o número de transações por hora. Observe que, nossa tabela de transações tem as datas das transações, logo, quando usamos o Count Values estamos obtendo o número médio de transações por hora de todos os dias. **OBS:** O Count Values tem um parâmetro opcional (numDays), mandamos ele para que ele possa calcular o número médio de transações por hora.

6 Banco

Os dados fictícios do projeto exemplo foram criados à partir do Python com a ajuda da biblioteca Faker. Basta o usuário determinar o número contas, transações e o número máximo de clientes para obter os arquivos csv correspondentes. As tabelas geradas são salvos como um banco de dados so SQLite e também em arquivos CSV.

O projeto exemplo é baseado em transações bancárias entre contas. Sendo assim, temos três tabelas principais:

- **Clientes**: Contém as informações pessoais de cada cliente. Cada registro possui um customer id único numerado por números naturais.
- **Contas**: Cada cliente pode possuir entre uma a três contas bancárias, sendo sorteados tipos de conta entre "corrente", "salário"e "poupança". Além disso, foi atribuído um saldo inicial aleatório entre R$ 0,00 e R$ 15.000,00, uma data de abertura (dentro do ano corrente) e um status (pendente, bloqueada ou desbloqueada). Cada conta também recebeu uma cidade de origem, selecionada aleatoriamente a partir de uma lista de capitais brasileiras.
- **Transações**: Cada transação está associada a contas remetentes e destinatárias. Os tipos de transação considerados foram depósito, retirada, transferência e pagamento. Cada transação (por questão de geração aleatório) recebe valores de R$ 10,00 até R$ 900,00. Cada movimentação incluiu informações como valor, localização, horário de início e término e data. Para simular atividades suspeitas, 2% das transações ocorreram em localidades diferentes da cidade associada à conta. Além disso, os dados de tempo são gerados de maneira contínua (ordenada).

As tabelas possuem os seguintes campos:

Coluna	**Descrição**	**Tipo**
customer_{id}	ID do cliente	INT
account_{id}	ID da conta	INT
current balance	Saldo atual da conta	FLOAT
account type	Tipo da conta	STRING
opening date	Data de abertura da conta	DATE
account status	Status da conta (Pendente, bloqueada, desbloqueada)	STRING
account location	Local de criação da conta	STRING

Tabela 1: Descrição das Colunas da Tabela de Contas

Coluna	**Descrição**	**Tipo**
customer_{id}	ID do cliente	INT
name	Nome do cliente	STRING
email	Email do cliente	STRING
address	Endereço do cliente	STRING
phone	Telefone do cliente	INT

Tabela 2: Descrição das Colunas da Tabela de Clientes

Obs: Vale ressaltar que os dados TIME e DATE para o C++ são na verdade Strings. Entretanto, seguem por convensão um padrão de escrita para realizar as operações necessárias. Além disso, os dados numéricos no geral foram gerados com distribuição uniforme.

Coluna	Descrição	Tipo
transaction_{id}	ID da Transação	INT
account_{id}	ID da conta de origem	INT
recipient_{id}	ID da conta de destino	INT
amount	Valor da transação	FLOAT
type	Depósito, retirada, transferência, pagamento	STRING
time start	Hora de início da transação	TIME
time end	Hora de fim da transação	TIME
date	Data da transação	DATE
location	Local onde foi realizada	STRING

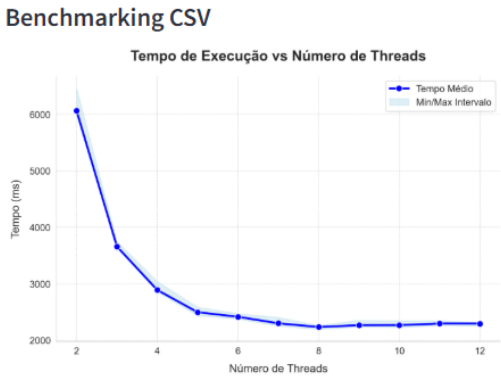
Tabela 3: Descrição das Colunas da Tabela de Transações

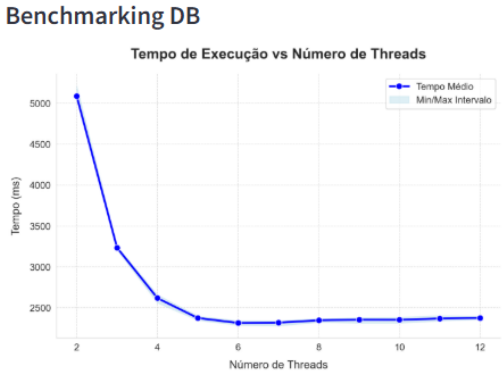
7 DashBoard

O DashBoard em questão foi feito em StreamLit, caso o usuário queira atualizar a página em virtude da mudança dos arquivos csv envolvidos para a geração dos dados, ele pode simplesmente clicar no botão Refresh.

Para visualizar todos os resultados gerados pelo sistema bancário, basta seguir o manual para rodar o home.py.

Por fim, foram registrados no DashBoard os tempos de execução no caso do sistema de extratores. Além disso, esses gráficos começam de duas threads (pelo menos uma dedicada para a leitura e outra para a escrita). A seguir, mostramos os resultados:





Para informações mais detalhadas do processo de criação, basta acessar: https://github.com/AlessandraBello/Comp_escalavel_a1