Fundação Getulio Vargas

Escola de Matemática Aplicada

Ciência de Dados e Inteligência Artificial

Almir Augusto Fonseca

Gabriel Jacinto Pereira

Gustavo Reis Rocha

Juliana Carvalho de Souza

Luan Rodrigues de Carvalho

T1 - Framework para Desenvolver Pipelines

Computação Escalável

Professor: Thiago Pinheiro de Araújo

Rio de Janeiro – RJ

Abril de 2024

1 Introdução

Este trabalho visa criar um framework para desenvolver pipelines de processamento de dados utilizando os mecanismos de execução de forma concorrente e paralela.

2 Mock

O simulador do sistema de *e-commerce*, construído em Python, simula realisticamente o fluxo de ações de vários usuários de acordo com um grafo de probabilidades. Entre as ações se incluem o login, navegação na página inicial, visualização de um produto, compra de um produto, logout. O sistema opera em ciclos para que os dados simulados sejam consistentes. Além disso, a cada ciclo é possível que novos usuários, produtos sejam criados, e as planilhas estoques e compras sejam também atualizadas. Ao final de cada ciclo produz o relatório gerando arquivos, que são formatados em `csv` e salvos em três pastas: **csv**, referente aos dados do Conta Verde, onde os arquivos são rotineiramente sobrescritos, **log** do DataCat, e **request** do Cadê Analytics, onde novos arquivos são adicionados a cada ciclo. As leituras são realizadas em intervalos de igual espaçamento nos dois primeiros tipos de dados e aleatoriamente nos dois últimos. O simulador é o primeiro passo no balanceamento de carga, já que é possível parametrizar o tamanho de um arquivo a ser processado.

3 Dataframe

A implementação da classe `DataFrame` é extensa e complexa, mas é essencial para a manipulação de dados tabulares. A classe é capaz de lidar com diferentes tipos de dados e fornece uma interface para realizar operações comuns em DataFrames, inspirada em bibliotecas populares como Pandas em Python.

Sua construção heterogênea surge a partir da manipulação de colunas homogêneas (`Series`) que, através do padrão de projeto `Interface`, fornecem acesso e funcionalidades sobre vetores parametrizados (`template<typename T>`) e retornos genéricos (`std::any`). Portanto, o container consiste num modelo que emprega polimorfismo e generics (templates) para permitir felxibilidade no armazenamento e manipulação de dados.

Dentre as funcionalidades implementadas, se destacam, por exemplo:

- **Métodos de manipulação de dados**: Adicionar e remover linhas (`addRow` e `dropRow`); adicionar e remover colunas (`addColumn`).
- **Métodos de consulta**: Obter o número de linhas e colunas (`getRowCount` e `getColumnCount`); obter o valor de uma célula (`getValueAt`).

- **Métodos de agregação**: Calcular a soma de uma coluna (`sum`); contar as ocorrências de cada valor em uma coluna (`valueCounts`).
- **Métodos de ordenação e filtragem**: Ordenar o DataFrame por uma coluna (`sortByColumn`); filtrar o DataFrame por uma coluna (`filterByColumn`).
- **Métodos utilitários**: Imprimir o DataFrame (`print` e `printColumnTypes`); criar cópias profundas de DataFrames (`deepCopy`).

Almejando a redução do acoplamento e a melhoria da coesão, a classe `DataFrame` foi projetada para ser extensível e flexível, permitindo a adição de novas funcionalidades e operações de forma modular. A implementação de métodos de manipulação de dados, agregação, ordenação e filtragem, além de outros métodos utilitários, concentra a definição dos métodos a serem utilizados pelos tratadores durante a execução da pipeline, garantindo a modularidade e a reusabilidade do código.

4 DataRepo

A classe `DataRepo` é responsável por fazer a leitura e escrita de arquivos `csv` e `txt`. Para obtenção de um *DataFrame* a partir de um arquivo, é necessário informar o caminho do arquivo e o separador utilizado, então ele realiza a conversão de cada coluna de dado para o seu respectivo tipo, enviando posteriormenete para a classe `DataFrame`. A diferença entre os métodos de leitura de arquivos `csv` e `txt` é que o primeiro exige que todas as linhas tenham o mesmo número de colunas, enquanto o segundo não impõe essa restrição, mas o restante do processo é o mesmo.

A classe `DataRepo` também é responsável por salvar um *DataFrame* em um arquivo `csv` ou `txt`. Para isso, é necessário informar o caminho do arquivo. O método de escrita em arquivo `csv` escreve o *DataFrame* em um arquivo `csv` separado por vírgulas, enquanto o método de escrita em arquivo `txt` escreve o *DataFrame* em um arquivo `txt` separado por tabulações.

5 ETL

Ao ativar, as *triggers* chamam métodos do ETL responsáveis por ler os dados do repositório e salvarem em um dataframe.

5.1 Pipeline

O processamento paralelo de dados é realizado usando um pool de threads que gerencia um conjunto de threads que extraem dados de vários arquivos, filtram e processa os dados simultaneamente. Existem filtros para separar dados por critérios, contadores para contar entradas e contadores de valores para contar ocorrências de

valores específicos. Após a conclusão de todas as tarefas, os resultados finais são processados e salvos em um arquivo.

5.2 Tratadores (Data Handler)

É responsável por ler dados de uma fila de entrada, processá-los e gravar o resultado em uma fila de saída. Isto permite o processamento paralelo de dados, uma vez que várias subclasses do DataHandler podem ser executadas simultaneamente, em threads diferentes. Dentre as subclasses se incluem:

- `FilterHandler`: filtra dados em um DataFrame com base em um nome de coluna e um valor de filtro.
- `CountLinesHandler`: conta o número de linhas (entradas) em um DataFrame.
- `ValueCountHandler`: conta o número de ocorrências de cada valor em uma coluna de um DataFrame.
- `JoinHandler`: combina dois DataFrames com base em uma coluna comum.
- `SortHandler`: ordena um DataFrame com base em uma coluna. Os dados podem ser ordenados em ordem crescente ou decrescente.
- `MergeAndSumHandler`: mescla dois DataFrames com base em uma coluna comum e soma os valores correspondentes dessa coluna específica.

5.3 Paralelismo

O paralelismo é alcançado por meio de um *pool* de *threads* que processam os dados de forma concorrente. Para isso, é utilizado o *framework* `ThreadPool` que gerencia um número fixo de *threads* que processam os dados de forma concorrente. O *pool* de *threads* é responsável por distribuir as tarefas de processamento de dados (*DataHandlers*) entre as *threads* disponíveis, garantindo que o processamento seja realizado de forma paralela.

Além dos tratadores de dados, o *framework* também conta com *tasks* que são responsáveis por realizar a junção dos resultados parciais obtidos por cada *DataHandler* e gerar o resultado final. O resultado final é requisitado por um *TimerTrigger* que, ao final de um intervalo de tempo, solicita o carregamento dos resultados em um arquivo (utilizando o *DataRepo*) e reinicia o ciclo de processamento.

6 Dashboard

A fim de produzir rapidamente um dashboard simples, escolhemos a linguagem Python, juntamente com o módulo *streamlit*, para exibir no formato de gráficos, os

resultados obtidos ao fim da pipeline. Como os arquivos de output são sobrescritos, o dashboard observa (a cada 3 segundos) o horário de modificação do arquivo para notar quando houve a sobrescrição/atualização dos dados a serem exibidos. Ao notar alterações em algum dos arquivos, todos as exibições são atualizadas:

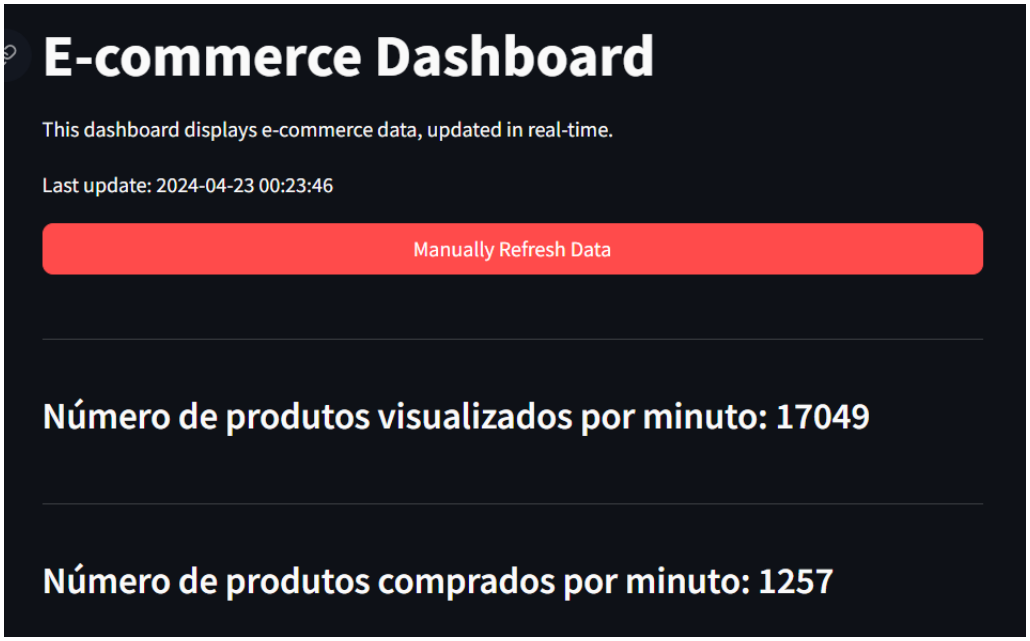

Figura 1: Cabeçalho do dashboard

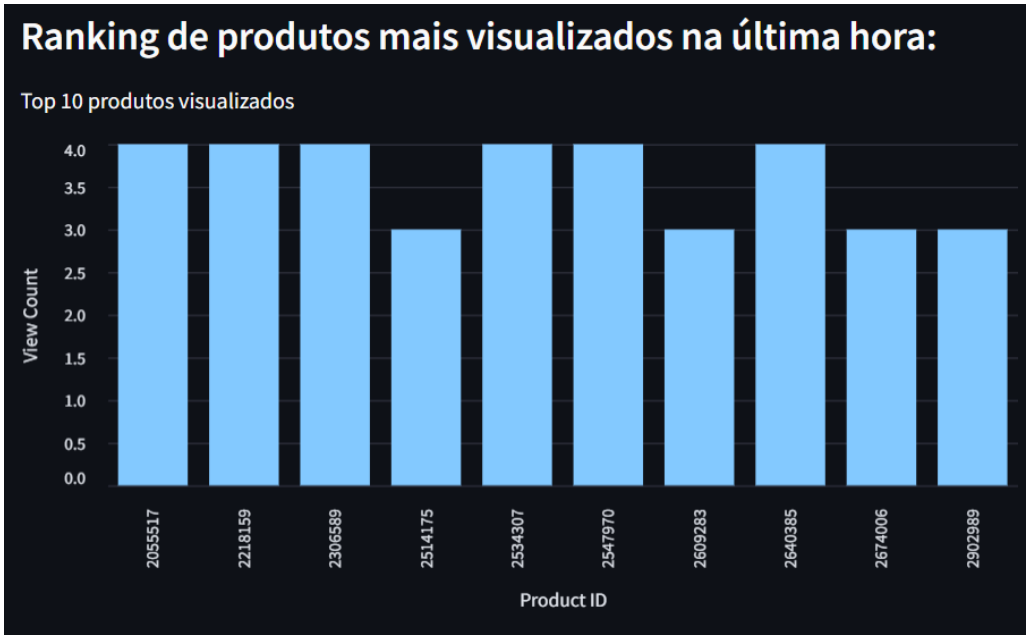

Figura 2: Visualizações do dashboard

Contudo, vale ressaltar que, diante das limitações da ferramenta *streamlit*, não foi possível ordenar as barras das visualizações, tornando as visualizações ranqueadas

um pouco informais quando se trata de rankings;