Análise comparativa do Trie Sort vs. Radix Sort para ordenação de texto

1st Felipe Estevanatto
UNIFESP
São Jose dos Campos, Brasil
felipe.estevanatto@unifesp.br

***Abstract*—Este relatório apresenta uma análise comparativa entre os algoritmos de ordenação Trie Sort e Radix Sort com *datasets* baseadas em texto no quesito performance e utilização de memória. O Objetivo da comparação é demonstrar como a escolha do algoritmo correto para cada tipo de conjunto de dados pode resultar em diferenças de performance de ordens de grandeza mesmo quando apenas pela sua análise de complexidade na notação *big O*, esses algoritmos possam ter a mesma categoria.**

***Index Terms*—Trie, Radix, Ordenação, Análise, Performance, Memória, Estrutura de dados, Python.**

I. Introdução

Quando o objetivo é a ordenação de grandes quantidades de dados, mesmo utilizando um computador moderno, ainda temos limitações para uso de memória ou tempo de execução. E de maneira intuitiva, podemos sempre adaptar nosso algoritmo de ordenação baseado no tipo de conjunto de dados a ser trabalhado, buscando mais eficiência para esse caso especifico, porém em outros casos dos qual o conjunto de dados é completamente aleatório, um algoritmo generalizado pode ser a escolha mais eficiente. Portanto dentro de um mesmo conjunto de dados, foram selecionadas diferentes colunas como índices para a ordenação, no objetivo de entender melhor essa escolha em relação a dois algoritmos que ocupam a mesma categoria de complexidade na notação *big O*, sendo eles o Radix Sort e o Trie Sort.

II. Trabalhos relacionados

Em "*A Fast Radix Sort*" de I.J.DAVIS [8], foi conduzido um experimento sobre a eficiência do algoritmo Radix Sort ser muito dependente do método utilizado para particionar as chaves, entretanto, em testes realizados até 65536 chaves, o Radix Sort sempre foi mais rápido que uma implementação em Quick Sort. O que mostra sua eficiência notável para valores aleatórios de tamanho moderado. Na publicação "*Selection of best sorting algorithm*" de Mishra, Aditya Dev e Garg, Deepak [6] é abordado que o problema de escolher o suposto "melhor algoritmo" de ordenação, é totalmente dependente do conhecimento sobre o conjunto de dados a ser utilizado, construindo então uma tabela que cita casos ideais de conjuntos de dados para cada tipo algoritmo de ordenação, que descreve o conjunto ideal para o Radix Sort como um alfabeto constante (alfabeto ordenado com poucos caracteres), e o conjunto ideal para um TAD(Tipo abstrato de dados) um set moderado de dados similares. O artigo *"Implementing collection of sets with trie: a stepping stone for performances ?*" [5] faz então uma análise de performance e consumo de memória de variações de ordenações por Trie (definido como uma árvore lexicográfica) em relação a mapas de Hash para diferentes conjuntos de dados, chegando ao resultado que para dados aleatórios, uma estrutura Trie, apesar de consumir muito mais memória que um *Hashmap*, em valores de:

$$N > 2^{20}$$

Essa estrutura tem uma inserção, busca e remoção menos custosa em processamento que uma HashMap imperfeito:

TABLE I
Experimental evaluation on Full examples using HashMap and TreeMap (N as 2N̂).

Full (n)	HashMap (ms)			TreeMap (ms)		
	put	get	remove	put	get	remove
12	78	101	90	274	90	28
13	134	149	155	639	51	30
14	265	138	157	792	42	33
15	484	94	96	981	45	38
16	1197	67	42	1275	87	71
17	2366	67	39	2251	91	74
18	5251	66	41	5692	41	65
19	12139	66	38	14928	47	52
20	35770	100	195	34078	65	65

A comparação frente ao Radix Sort foi escolhida baseada no artigo "Efficient Trie-Based Sorting of Large Sets of Strings" [2] de 2003 que sugere um algoritmo Trie Sort em conjunto com outro algoritmo de seleção de prefixos para ordenar grandes quantidades de dados. A ideia é similar a da boa escolha de um pivô no Quicksort ou *gap* no ShellSort. Porém quando se trata de strings, principalmente texto escrito por humanos, os padrões repetitivos de escrita permite o Trie Sort ser muito mais eficiente nessa escala.

É esperado que os resultados encontrados sejam semelhantes apesar do ambiente computacional e linguagens escolhidas neste trabalho sejam completamente diferentes.

III. Metodologia

A. Dados utilizados

Os dados utilizados para a ordenação são os dados anonimizados do banco de dados de E-commerce brasileiro da Olist

de 2016 até 2018, liberado pela própria empresa e disponível no Kaggle [1]. Das tabelas disponíveis podemos fazer análises dos arquivos CSV disponíveis e relacionar os resultador por meio do diagrama do banco de dados da Olist liberado junto aos dados [Fig. 1]

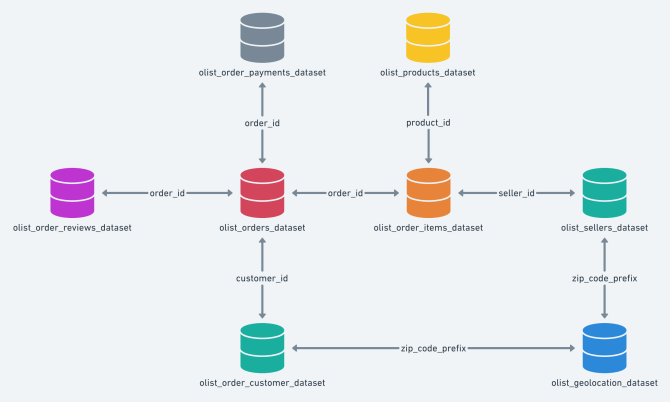

Fig. 1. Diagrama do banco de dados da Olist.

O *dataset* da Olist é separado em 9 arquivos, para cada tabela do banco e suas relações. Entretanto, os seu dados foram exportados de uma forma bruta do banco de dados original, portanto não são normalizados para uma padrão especifico, alguns campos são nulos ou vazios, e outros contem valores distintos dentro de um campo supostamente padronizado, então, em alguns casos foi preciso normalizar os dados da coluna à ser ordenada, por motivos simplificação de casos extremos para os algoritmos de ordenação escolhidos.

B. Linguagem Utilizada

A linguagem escolhida foi o Python, que apesar de não ser a linguagem mais performática, tem um código mais legível a primeiro momento, e com o tamanho de dados grande o suficiente, ainda teremos um resultado bem dependente da combinação entre o tipo de algoritmo e o conjunto de dados

C. Código principal do Benchmark

Os datasets, todo o código utilizado para ordenação, normalização de valores e geração de gráficos utilizados neste artigo podem ser encontrados no repositório do GitHub [4]. A construção do *framework* de *benchmark*, assim como a validação de I/O de arquivos CSV e implementação dos algoritmos de ordenação com esses valores foi feita com o auxilio do chatGPT [7]

```python
import os
import cProfile
import csv
import io
import pstats
from trie_sort import trie_sort
from radix_sort import radix_sort
from memory_profiler import memory_usage

# Set the current working directory
os.chdir(os.path.dirname(__file__))

def read_csv(file_path):
    with open(file_path, 'r') as csv_file:
        return list(csv.reader(csv_file))

def average(stats, count):
    stats.total_calls /= count
    stats.prim_calls /= count
    stats.total_tt /= count

    for func, source in stats.stats.items():
        cc, nc, tt, ct, callers = source
        stats.stats[func] = (cc/count, nc/count, tt/count, ct/count, callers)

    return stats

def best_of_profillings(target_profile_function, count, *args):
    output_stream = io.StringIO()
    profiller_status = pstats.Stats(stream=output_stream)
    mem_usage = []

    for index in range(count):
        profiller = cProfile.Profile()
        profiller.enable()

        mem_usage.append(memory_usage((target_profile_function, args)))

        profiller.disable()
        profiller_status.add(profiller)

        print('Profiled', '%.3f' % profiller_status.total_tt, 'seconds at', index,
              'for', target_profile_function.__name__, flush=True)

    average(profiller_status, count)
    profiller_status.sort_stats("time")
    profiller_status.print_stats()

    avg_mem_usage = [sum(x)/len(x) for x in zip(*mem_usage)]
    print(f"Average memory usage for {target_profile_function.__name__}: {avg_mem_usage} MiB")

    return "\nProfile results for %s\n%s" % (target_profile_function.__name__, output_stream.getvalue())

if __name__ == "__main__":
    # Read the CSV file
    data = read_csv('datasets/olist_geolocation_dataset_normalized.csv')

    # Profile and measure memory usage during sorting
    heavy_lifting_result = best_of_profillings(radix_sort, 10, data, 3)
    print("Radix sort\n" + heavy_lifting_result)

    # Profile and measure memory usage during trie sort
    heavy_lifting_result = best_of_profillings(trie_sort, 10, data, 3)
    print("Trie sort\n" + heavy_lifting_result)
```

D. Código do Radix Sort

O Radix Sort foi escolhido pois é de simples implementação e tem uma performance aceitável para intervalos de valores pequenos (a-Z), se tornando uma ótima alternativa

para ordenação de strings idealmente de tamanhos iguais. A implementação do Radix Sort funciona ordenando dados com base nos caracteres de uma coluna específica inteira. Ele começa pela posição menos significativa (último caractere) e vai para a mais significativa, utilizando uma abordagem de ”distribuição” em baldes (buckets). Para cada posição de caractere, ele organiza os elementos de acordo com o valor ASCII do caractere naquela posição. Se uma string for mais curta do que a posição atual, o algoritmo usa um valor de preenchimento (0, no caso) para garantir que todas as strings sejam comparáveis. Esse processo de distribuição e reconstrução da lista se repete até que todos os caracteres da coluna tenham sido considerados.

```python
def radix_sort(data, sort_column):
    # Get the maximum length of the strings in the specified column
    max_length = max(len(row[sort_column]) for row in data)

    # Sort from the least significant character to the most significant character
    for i in range(max_length - 1, -1, -1):
        # Initialize 256 buckets for each ASCII character
        buckets = [[] for _ in range(256)]

        # Distribute the rows into buckets based on the current character index
        for row in data:
            if i < len(row[sort_column]):
                char_index = ord(row[sort_column][i])
            else:
                char_index = 0  # Use 0 for padding if the index is out of range
            buckets[char_index].append(row)

        # Flatten the buckets back into the data list
        data = []
        for bucket in buckets:
            data.extend(bucket)

    return data
```

E. Código do Trie Sort

O Trie Sort é um algoritmo de ordenação que utiliza a estrutura de dados chamada Trie, que como mencionado anteriormente, é uma árvore de prefixos. Cada nó na árvore representa um caractere, e as palavras são inseridas de forma que compartilham prefixos comuns, tornando a Trie eficiente em termos de espaço e tempo ao manipular grandes volumes de dados com prefixos similares. No caso do algoritmo apresentado, a Trie armazena as strings de uma coluna específica e organiza os dados em uma estrutura hierárquica onde cada nível da árvore corresponde a um caractere da string. Quando um dado é inserido, o algoritmo percorre cada caractere da string e cria um caminho na árvore, criando nós para os caracteres ausentes. No final da string, um nó é marcado com uma propriedade para ser um fim da palavra e os dados associados a essa palavra são armazenados nesse nó. Após a inserção de todos os dados, a ordenação é realizada por uma busca em profundidade (DFS), onde as palavras são recuperadas em ordem lexicográfica ao percorrer os nós da Trie.

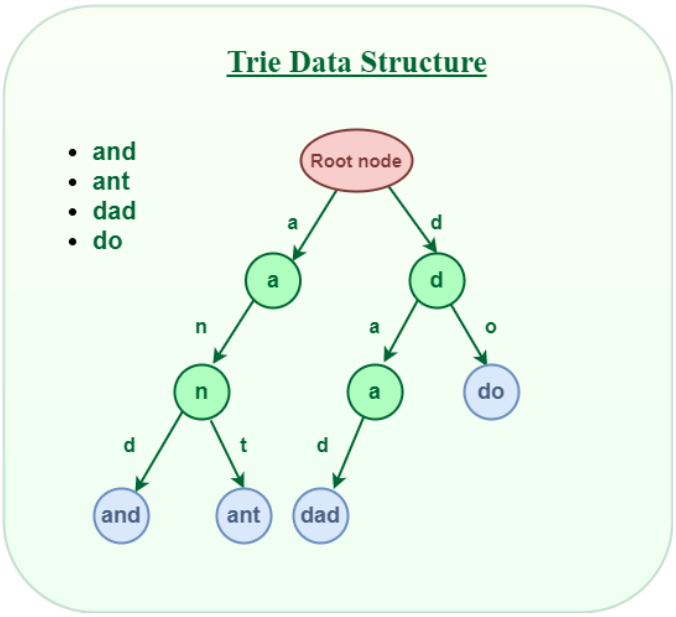

Fig. 2. Exemplificação de uma estrutura Trie [3].

Esse algoritmo é particularmente eficiente para ordenar strings, inclusive de tamanhos variados, pois a estrutura Trie permite que prefixos compartilhados sejam armazenados de forma compacta, otimizando a performance de inserção e busca. Além disso, o Trie Sort mantém a ordem de inserção para strings com prefixos iguais, o que é uma característica importante em alguns cenários de ordenação.

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False
        self.data = []  # To store the count of occurrences

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, key, row):
        node = self.root
        for char in key:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True
        node.data.append(row)

    def _traverse_dfs(self, node, prefix, result):
        if node.is_end_of_word:
            for row in node.data:
                result.append((prefix, row))
        for char in sorted(node.children.keys()):
            self._traverse_dfs(node.children[char], prefix + char, result)

    def sort(self):
        result = []
        self._traverse_dfs(self.root, "", result)
        return result
```

```python

def trie_sort(data, sort_column):
    trie = Trie()
    for row in data:
        trie.insert(row[sort_column], row)

    # Retrieve the sorted data
    sorted_data = trie.sort()
    return [row for _, row in sorted_data]
```

IV. Análise Experimental

A. Conjunto de dados

- O arquivo `olist_geolocation_dataset.csv` que contem uma localização generalizada de todos os endereços cadastrados pelos clientes e vendedores na plataforma, totalizando cerca de 1.000.000 valores. A coluna a ser utilizada para ordenação será a coluna ”geolocation_city”, o arquivo original normalizado tem cerca de 68MB.
- O arquivo `olist_order_payments_dataset.csv` que contem os valores, tipo e parcelas de todos os pagamentos relacionados a um pedido, totalizando cerca de 100.000 valores. A coluna a ser utilizada para ordenação será a coluna ”order_id”, que é um UUID(Identificador Único Universal).

B. Configuração do Algoritmo e Ambiente Computacional

A versão utilizada do Python é a 3.12.7, e os teste foram executados em um computador com as seguintes configurações:

- Processador AMD Ryzen 7 5700X 8/16
- Memória RAM DDR4 3600Mhz 4x8GB T-Force
- SSD NVMe PCIe4.0 2TB Netac NV7000

O gerenciamento do arquivo CSV, calculo do tempo de execução e memória utilizada foram feitos utilizando as bibliotecas nativas do Python para evitar influência de bibliotecas externas utilizando ou não funções compiladas de PyC em cada método de ordenação.

C. Resultados e Discussão

O primeiro resultado veio de simplesmente ordenar o primeiro conjunto de dados do arquivo `olist_geolocation_dataset.csv` normalizado por meio das cidades de cada geolocalização, o resultado experado seria de uma grande vantagem para o Trie Sort, que trabalhar literalmente com uma estrutura de prefixos de palavras, que se torna muito útil quando N cresce muito, porém a variação de N se mantem limitada (quantidade de cidades brasileiras).

TABLE II
RESULTADOS MÉDIOS DE 10 EXECUÇÕES DO CPROFILE PARA O `RADIX_SORT` E `TRIE_SORT` PARA N DE 1 MILHÃO.

Metrica	radix_sort	trie_sort
Chamadas de função	84,487,756.9	3,140,654.0
Chamadas primitivas de função	84,487,752	3,105,686
Tempo de execução (s)	17.243	2.344
Uso médio de memória (MiB)	453.20859375	508.976953125

Outra ordenação com menos variações que pode ser feita no mesmo set de dados seria a ordenação lexicográfica dos estados brasileiros, com apenas 2 caracteres e uma variação ainda menor de letras:

TABLE III
RESULTADOS MÉDIOS DE 10 EXECUÇÕES DO CPROFILE PARA O `RADIX_SORT` E `TRIE_SORT` PARA N DE 1 MILHÃO COM M = 2.

Metrica	radix_sort	trie_sort
Chamadas de função	38,011,073.9	3,001,002.0
Chamadas primitivas de função	38,011,069	3,000,947
Tempo de execução (s)	7.813	1.625
Uso médio de memória (MiB)	449.24609375	500.13671875

Com ambos os resultados é possível observar que a vantagem de performance do Trie Sort continua grande, porém com uma proporção menor, e o consumo de memória continua bem maior que a ordenação por Radix Sort. O alto consumo de memória no geral para ambos os algoritmos se dá a maneira como o código é interpretado pelo ambiente Python na máquina em conjunto do tamanho do conjunto de dados a ser ordenado (cerca de 68MB normalizado).

E a partir dos dados ordenados, podemos fazer gráficos mostrando conclusões que podemos extrair a partir desses dados. Um gráfico simples seria a contagem de ocorrência por cidade, porém como o dataset incluí o Brasil todo, há muitos valores possíveis, que fora agregados então como *Others* na Fig. 2.

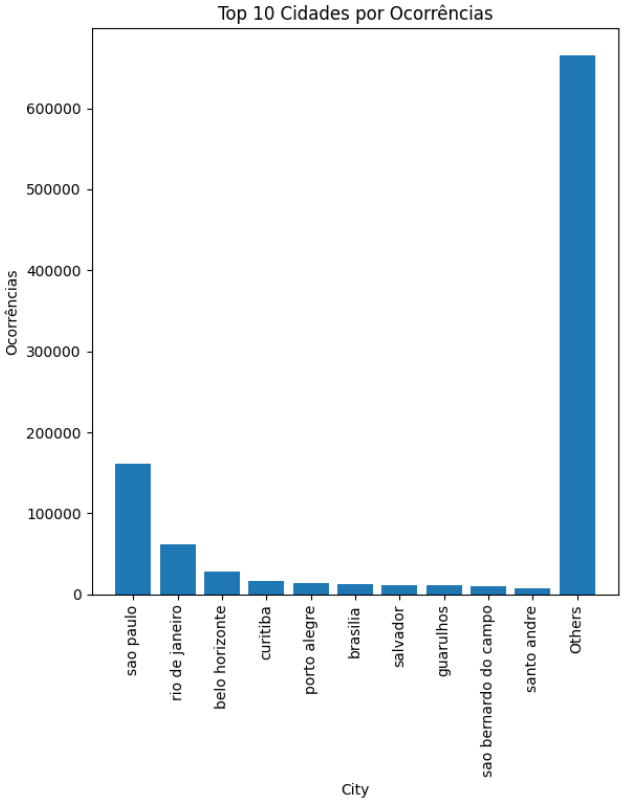

Fig. 3. Top 10 Cidades com mais endereços cadastrados.

O resultado do gráfico de ocorrência de cidades [Fig. 2] mostra um resultado bem intuitivo, as cidades mais populosas,

tendem a ter mais usuários da plataforma, portanto mais endereços na região, porém é interessante apontar que mais da metade dos endereços não pertence à essas capitais. Em uma análise da quantidade de vezes que um estado é mencionado, e possível observar melhor como a população de cada estado é de certa forma proporcional a quantidade de endereços cadastrados no estado. Ou seja, a população de usuários do sistema, é muito representativa para o todo de cada estado.

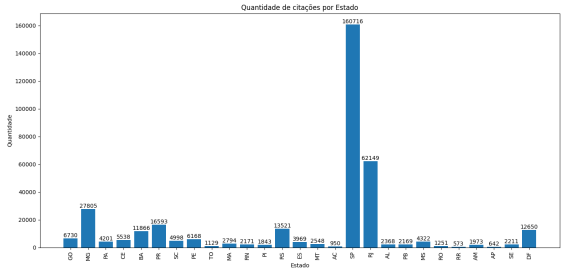

Fig. 4. Quantidade de citações de cada estado.

O ultimo caso seria o do arquivo `olist_order_payments_dataset.csv`, ordenado pelo campo de "order_id", representado pela Tabela 4.

TABLE IV
RESULTADOS MÉDIOS DE 10 EXECUÇÕES DO CPROFILE PARA O `RADIX_SORT` E `TRIE_SORT` PARA 100 MIL UUIDS.

Metrica	**radix_sort**	**trie_sort**
Chamadas de função	10,189,568.9	11,651,527
Chamadas primitivas de função	10,189,564	8,816,633
Tempo de execução (s)	2.560	7.410
Uso médio de memória (MiB)	72.982	720.204

O resultado dessa análise é interessante para o algoritmo Radix Sort pois já tem uma quantidade de dados muito menor que a primeira análise, e o fato dos valores serem strings completamente aleatórias é uma grande desvantagem para o Trie Sort, que consome muita memória e ainda sim tem uma performance muito inferior pois quase todo valor é um caso extremo, ou seja, não há nenhum prefixo em comum entre os valores. Resultando em uma performance cerca de 3x pior, consumindo 10x mais memória.

V. CONCLUSÃO

Como demonstrado inicialmente pelos trabalhos mencionados na seção II, o conjunto de dados a ser ordenado é um grande fator para a escolha e eficiência final dos algoritmos de ordenação. E o Trie Sort em especifico tem muito potencial para ordenação de strings não aleatórias no qual a restrição de memória não é um problema, e uma vez que a sua estrutura de dados é construída, trabalhar com ela se torna tão eficiente quando utilizar um HashMap. E com base no dataset utilizado, é possível observar de maneira fácil casos em que em relação aos endereços de todos os clientes e vendedores do dataset da Olist, é possível relacionar o publico alvo de maneira quase proporcional a densidade populacional. E com os outros datasets disponíveis, ligar as cidades e estados chave com os locais com as vendas de maior ticket, ou com uma categoria especifica de produto se destacando se torna simples.

REFERÊNCIAS

[1] KAGGLE. Brazilian E-Commerce Public Dataset by Olist. Disponível em: https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce. Acesso em: 26 nov. 2024.

[2] Sinha, Ranjan & Zobel, Justin. (2003). Efficient Trie-Based Sorting of Large Sets of Strings. Disponível em: https://www.researchgate.net/publication/2556055_Efficient_Trie-Based_Sorting_of_Large_Sets_of_Strings. Acesso em: 26 nov. 2024.

[3] GEEKSFORGEEKS. Trie — (Insert and Search). Disponível em: https://www.geeksforgeeks.org/trie-insert-and-search/. Acesso em: 26 nov. 2024.

[4] FELIPEESTEVANATTO. GitHub - FelipeEstevanatto/Trab-AED2: Códigos para o Relatórios de AED2. Disponível em: https://github.com/FelipeEstevanatto/Trab-AED2. Acesso em: 26 nov. 2024.

[5] Simon Bachelard, Olivier Raynaud, Yoan Renaud. Implementing collection of sets with trie: a stepping stone for performances ?. 2006. ffhal-00640978f. Disponível em: https://hal.science/hal-00640978v1/document. Acesso em: 26 nov. 2024.

[6] MISHRA, Aditya Dev; GARG, Deepak. Selection of best sorting algorithm. International Journal of intelligent information Processing, v. 2, n. 2, p. 363-368, 2008. Disponível em: https://www.academia.edu/download/28569137/Selection_of_best_sorting_algorithm.pdf. Acesso em: 26 nov. 2024.

[7] OpenAI, "ChatGPT," disponível em https://chat.openai.com. Acesso em: 26 nov. 2024.

[8] I. J. Davis, A Fast Radix Sort, The Computer Journal, Volume 35, Issue 6, December 1992, Pages 636–642, Disponível em: https://academic.oup.com/comjnl/article/35/6/636/352883. Acesso em: 26 nov. 2024.