Universidade Federal Rural do Semi-Árido – UFERSA

Campus Multidisciplinar de Pau dos Ferros – CMPF

Disciplina: Lab. de Algoritmos e Estrutura de Dados II

Docente: Kennedy Reurison Lopes

Discentes: Cláudio Felipe Lopes da Silva
Felipe Andrade da Silva

RELATÓRIO

(Verificador de Duplicatas)

Introdução

O projeto tem como objetivo desenvolver um sistema capaz de identificar e sinalizar emails duplicados em listas grandes, a partir de arquivos CSV, utilizando diferentes técnicas de estrutura de dados para comparação de eficiência e escalabilidade. Trata-se de um trabalho acadêmico para a disciplina Estrutura de Dados II, sendo implementado em C.

Esse tipo de ferramenta é útil para empresas e sistemas que precisam garantir a unicidade de emails em bases de dados, como sistemas de cadastro e marketing, evitando redundância e garantindo integridade.

Estrutura Geral do Programa

O programa é dividido em três camadas principais:

- **Camada de Entrada:** leitura de arquivo CSV, validação e armazenamento dos emails em memória.
- **Camada de Processamento:** três métodos distintos para identificar duplicatas:
  - Filtro Bloom + Tabela Hash
  - Tabela Hash simples
  - Busca Linear
- **Camada de Saída:** exibição dos resultados e relatórios comparativos dos tempos de execução, via terminal ou interface gráfica.

O programa é modularizado em arquivos `.c` e `.h` para separar responsabilidades e facilitar manutenção.

Processos Detalhados

Leitura e Validação de Emails

O arquivo CSV é aberto no modo leitura (`fopen`).

A primeira linha é ignorada, pois geralmente contém o cabeçalho.

Cada linha subsequente é lida e tokenizada com `strtok` usando vírgula como separador.

O programa extrai o primeiro token, que corresponde ao email, e armazena numa matriz `emails[MAX_EMAILS][MAX_EMAIL_LEN]`.

Para garantir comparações consistentes, o email é convertido para letras minúsculas, utilizando a função `to_lower_str`.

A função `is_valid_email` faz uma validação simples, aceitando somente caracteres alfanuméricos, hífen, sublinhado, ponto e obrigatoriamente um caractere `'@'`.

Emails inválidos são ignorados, e o processo continua até o final do arquivo ou atingir o limite máximo definido (1 milhão de emails).

Essa etapa garantiu evitar erros futuros na comparação por entradas malformadas.

Implementação do Filtro Bloom

O filtro Bloom é uma estrutura probabilística que permite testar se um elemento possivelmente está presente numa coleção, com baixa probabilidade de falsos negativos.

Foi implementado um vetor de bytes de 125.000 (1 milhão de bits) para armazenar o filtro.

Utilizamos 3 funções hash diferentes baseadas na variação do algoritmo djb2, para definir múltiplos bits para cada email.

Ao adicionar um email ao filtro (`bloom_add`), os bits calculados pelas 3 funções hash são marcados.

Ao verificar se um email está presente (`bloom_possibly_contains`), todos os 3 bits devem estar marcados; se algum não estiver, o email com certeza não está.

Essa estratégia reduz drasticamente o número de buscas na tabela hash, acelerando o processo de identificação de duplicatas em grandes volumes.

Estrutura da Tabela Hash

A tabela hash é implementada como um array de ponteiros para listas encadeadas (`Node* hashTable[HASH_TABLE_SIZE]`), permitindo tratamento eficiente de colisões.

O tamanho da tabela foi definido como um número primo (10007) para minimizar colisões.

Cada nó da lista encadeada contém uma string de email e um ponteiro para o próximo nó.

A função `hash_func` calcula o índice da tabela baseado no hash djb2 aplicado ao email em minúsculas.

Para busca (`search_hash`), percorremos a lista encadeada do índice correspondente até encontrar o email ou chegar ao final.

Para inserção (`insert_hash`), criamos um novo nó no início da lista encadeada, evitando percorrer toda a lista.

É fundamental liberar a memória após o uso para evitar vazamentos, feito em `free_hash_table`.

Busca Linear

Método básico de busca sequencial, percorre um array contendo emails já vistos para verificar duplicatas.

Inicialmente, o armazenamento do conjunto de emails verificados foi feito com array estático, o que causava falhas quando o volume de dados aumentou.

Para resolver, foi implementada alocação dinâmica (`malloc`) para criar um array que cresce conforme o total de emails, permitindo flexibilidade e evitando estouro de memória.

Embora simples, esse método é computacionalmente custoso ($O(n^2)$) para listas grandes, servindo principalmente para fins de comparação.

Medição e Relatórios de Desempenho

Utiliza-se a função `clock()` para medir o tempo de execução de cada método, capturando o tempo inicial e final.

Os tempos são armazenados em variáveis globais e exibidos ao usuário.

O relatório compara o desempenho dos métodos, apresentando percentuais de ganho ou

perda de eficiência, facilitando a análise.

Essa parte foi fundamental para validar a eficiência da combinação filtro Bloom + tabela hash, mostrando sua vantagem clara.

Interface de Usuário

Interface de terminal: oferece menu com opções numeradas para carregar arquivo CSV, executar cada método, exibir relatório e sair.

Interface gráfica (Windows): criada com WinAPI, possui botões para cada função, caixa de texto para mostrar resultados e seleção de arquivos via diálogo.

A interface gráfica facilita o uso por usuários não técnicos e permite interação mais visual e direta.

Durante o desenvolvimento, foi necessário aprender a manipular elementos gráficos, eventos de clique e atualizações assíncronas na GUI.

Principais Desafios e Soluções

Gerenciamento de Memória

Problemas iniciais com arrays estáticos que não suportavam listas muito grandes.

Implementação de alocação dinâmica para a busca linear, usando `malloc` e `free`, para evitar estouros.

Cuidados especiais na liberação das listas encadeadas da tabela hash para evitar vazamento.

Validação e Padronização dos Dados

Necessidade de padronizar emails para letras minúsculas para evitar que “Exemplo@dominio.com” e “exemplo@dominio.com” sejam tratados como diferentes.

Validação simples porém eficiente para garantir que somente emails bem formados sejam processados, evitando erros e falsos positivos.

Implementação do Filtro Bloom

Entender e aplicar conceitos de filtro Bloom, criando múltiplas funções hash para reduzir falsos positivos.

Garantir que o filtro estivesse corretamente dimensionado para o volume de dados.

Lidar com a modulação dos bits para evitar acesso fora dos limites do vetor.

Modularização do Código

Separação clara entre interface, lógica de negócio e manipulação de dados.

Uso de arquivos `.c` e `.h` para organizar funções e estruturas, facilitando manutenção e extensibilidade.

Documentação e comentários detalhados para explicar funções e variáveis.

Interface Gráfica

Aprender e implementar programação Windows nativa.

Garantir atualização da interface em tempo real sem travar o programa.

Tratar eventos e erros de forma amigável.

Resultados Obtidos

O método Filtro Bloom + Tabela Hash mostrou-se o mais eficiente, reduzindo o tempo de verificação em comparação à busca linear em até dezenas de vezes, especialmente em

grandes volumes.

A busca linear, apesar da simplicidade, foi útil como baseline para comparação, confirmando a vantagem da estrutura hash.

O filtro Bloom, por ser probabilístico, acelerou as verificações evitando buscas desnecessárias na tabela hash, sem impactar a precisão (não houve falsos negativos).

O sistema mostrou-se robusto para até 1 milhão de emails, com interface amigável e mensagens claras para o usuário.

Possíveis Melhorias Futuras

Implementar geração de arquivo CSV contendo apenas emails únicos para facilitar limpeza de bases.

Criar função para mesclar múltiplos arquivos CSV com remoção automática de duplicatas.

Otimizar o filtro Bloom e tabela hash para melhor uso de memória e desempenho.

Adicionar suporte a emails com caracteres especiais conforme padrões modernos.

Aprimorar interface gráfica com gráficos de desempenho e opções avançadas de filtro.

Conclusão

Este projeto demonstrou a importância e a eficiência do uso de estruturas de dados adequadas para resolver problemas práticos, como a identificação de duplicatas em grandes volumes de dados. A implementação combinada do filtro Bloom com a tabela hash mostrou-se eficaz ao reduzir significativamente o tempo de processamento, mantendo a precisão necessária para o contexto.

Além disso, a modularização do código e o desenvolvimento de uma interface gráfica

tornaram o sistema mais robusto e acessível, permitindo que usuários com diferentes níveis técnicos possam utilizar a ferramenta facilmente.

Os desafios enfrentados, especialmente relacionados à gestão de memória e validação dos dados, foram superados com soluções eficientes que garantiram a escalabilidade do programa para até um milhão de emails.

Por fim, o projeto abre portas para futuras melhorias, demonstrando como conceitos teóricos aprendidos em sala de aula podem ser aplicados para resolver problemas reais, promovendo aprendizado prático e desenvolvimento de habilidades essenciais em programação e engenharia de software.