Aplicando Hashing Extensível para Gerenciamento de Arquivos em um Sistema de Cadastro: análise do desempenho do software entre máquinas e unidades de armazenamento distintas.

Bruno Kelsen Vilarino de Almeida [1]
Dayane Gabriela Santos Cordeiro [2]
Samuel Vieira Lobo Chiodi [3]

[1] Bacharel em Geografia e Bacharelando em Engenharia da Computação - Instituto de Ciências Exatas e Informática – Pontifícia Universidade Católica (PUC Minas) - Belo Horizonte, MG - Brasil.

[2] Bacharelanda em Engenharia da Computação - Instituto de Ciências Exatas e Informática – Pontifícia Universidade Católica (PUC Minas) - Belo Horizonte, MG - Brasil.

[3] Bacharelando em Engenharia da Computação - Instituto de Ciências Exatas e Informática – Pontifícia Universidade Católica (PUC Minas) - Belo Horizonte, MG - Brasil.

bkelsen.engcomput@gmail.com
dayane.cordeirogs@gmail.com
samchiodi@gmail.com

Abstract. This work describes the implementation and behavior of an algorithm based on the dynamic hash technique, involving the description of the elements used in order to meet the programming requirements, simulation of data manipulation at different concentration scales and performance analysis according to runtime, physical computing resources and data size. Regarding the assembly of the algorithm developed in the C++ language, it clarifies the logic used based on the pre-existing procedure for indexing files by mathematical functions with dynamic creation of buckets. In terms of the simulations performed, it presents the flash memory used and describes the hardware architecture of the two machines used in the simulation containing Intel Core i7 and Intel Core i5 processors. It also exposes the data collection raised in each of the events, handling one thousand, ten thousand, one hundred thousand and one million records. Finally, the information obtained in the previous topic is analyzed using graphics that show performance comparisons between the two machines, the amount of information handled and total stage of action.

Resumo. Este trabalho descreve a implementação e o comportamento de um algoritmo baseado na técnica de hash dinâmico, envolvendo a descrição dos elementos utilizados a fim de atender aos requisitos de programação, simulação de manipulação de dados em diferentes escalas de concentração e análise de performance de acordo com o tempo de execução, recursos de físicos computacionais e tamanho dos dados. Em relação à montagem do algoritmo desenvolvido na linguagem C++, esclarece a lógica utilizada com base no procedimento preexistente para indexação de arquivos por funções matemáticas com criação dinâmica de buckets. Em termos das simulações realizadas, apresenta a memória flash utilizada e descreve a arquitetura do hardware das duas máquinas utilizadas na simulação contendo processadores Intel Core i7 e Intel Core i5. Expõe também a coleta de dados levantada em cada um dos eventos, manuseando mil, dez mil, cem mil e um milhão de registros. Por fim, as informações obtidas no tópico anterior são analisadas por meio de gráficos que exibem comparações performáticas entre as duas máquinas, a quantidade de informações manejadas e etapa total de atuação.

1. Introdução

O grande acúmulo de dados gerados pela computação sempre foi um problema a ser resolvido, esse grande volume de dados encontra-se principalmente armazenado em HDDs, SSDs e fitas. Faz-se então, necessário o uso de algoritmos que possibilitem acessar esses dados de maneira eficiente.

Este trabalho prático tem o objetivo implementar e avaliar o método de acesso em memória secundária e sua eficiência. Para isso foi implementado um sistema de manipulação e organização dos dados de cadastro de prontuários para uma empresa de planos de saúde.

Esse sistema gerencia dados por meio do uso de arquivos em memória secundária (disco rígido) cuja a organização é sequencial indexável, sendo os índices baseado na abordagem dinâmica do hashing extensível. O sistema dá aos usuários permissões básicas para criar, inserir, editar, excluir, imprimir e fazer buscas de acordo com o requisitado.

Proposto por Ronald, Nievergelt, Pippenger & Strong (1979) o hashing extensível é uma técnica de acesso na qual o usuário tem a garantia de não mais do que duas falhas de página para localizar os dados associados a um determinado identificador exclusivo ou chave. Ao contrário do hashing convencional, o hashing extensível tem uma estrutura dinâmica que aumenta e diminui normalmente à medida que o banco de dados aumenta e diminui. Essa abordagem simultaneamente resolve o problema de fazer tabelas hash que são extensíveis e de fazer árvores de pesquisa raiz que são balanceadas.

Portanto, foi estudado, por análise e simulação, o desempenho do hashing extensível no sistema de gerenciamento de arquivos implementado neste trabalho. Os resultados aqui apresentados buscam evidenciar que o hashing extensível fornece uma alternativa atraente para com outros métodos de acesso em memória secundária como, por exemplo, as árvores B e B+.

2. Objetivo e Metodologia

Esta atividade prática se propõe a realizar a implementação de um sistema básico de gerenciamento de banco de dados, a explicação da funcionalidade desse sistema e a realização de testes, onde se busca avaliar o desempenho do sistema nos processo de inserção, edição, pesquisa e recuperação da informação em memória secundária. Durante o processo de execução do programa serão coletados os dados necessários para este estudo.

Sendo executado três testes de tempo no programa em duas máquinas distintas, a simulação analisará a manipulação de dados aleatórios, indexados e sem repetição, com quatro diferentes volumes de registro, 1 mil, 10 mil, 100 mil e 1 milhão, sendo que com 1 milhão de registros o arquivo-mestre tem tamanho superior a 1 GB, para obter uma significativa quantidade de dados que facilite o entendimento da análise posterior deste programa.

3. Modelagem e Funcionalidades Implementadas

O desenvolvimento do sistema de gerenciamento de banco de dados para uma empresa de planos de saúde necessariamente inclui as funções de criar arquivo, inserir, editar, excluir registros, imprimir todos registros no arquivos e pesquisar um determinado registro obtendo também as métricas de performance do sistema ao executar a pesquisa. Os registros são indexados e gerenciados utilizando a técnica de hashing extensível (Figura 1).

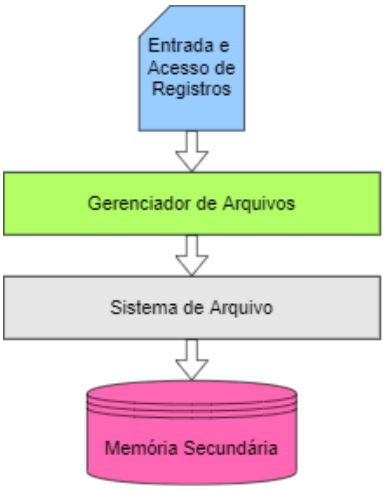


Figura 1. Organização do sistema de gerenciamento

Ao iniciar o programa o menu do sistema apresenta as seguintes opções onde o usuário deve digitar o número correspondente:

"0 - Exit"
"1 - Create file system"
"2 - Add"
"3 - Edit"
"4 - Remove"
"5 - Print file's contents"
"6 - Run a simulation"

As funções implementadas possuem as seguintes finalidades:

Create file system: cria um novo sistema de arquivos constituído por um diretório, um índice e um arquivo-mestre.

Add: insere um novo registro através da entrada dos dados nos campos, exceto, o campo de anotações, os campos são:

"CPF: "
"Name: "
"Birthdate: --/--/--"
"Gender: "
"Notes: "

O tempo decorrido no processo é exibido na tela após o término da inserção dos dados.

Edit: executa uma busca no arquivo a partir de um determinado CPF e permite a edição dos campos deste registro. O tempo decorrido no processo de pesquisa e recuperação do registro é exibido na tela após o término da edição dos dados.

Remove: exclui um determinado registro do arquivo a partir de um CPF. O procedimento de remoção é sempre lógico, zerando-se o CPF do bucket correspondente e endereço para o arquivo-mestre recebe um flag de valor 1. Observação: o espaço do registro do arquivo-mestre não é excluído e pode ser reutilizado em uma próxima inclusão.

Print file's contents: exibe na tela o atual diretório e todo o conteúdo dos registros armazenados no arquivo-mestre.

Run a simulation: criar um sistema de arquivos onde se define o tamanho em Bytes do arquivo, a quantidade de buckets por página e o número máximo do conjunto de CPF sem repetições que serão inseridos neste arquivo inicialmente vazio. Em seguida, as chaves são aleatoriamente pesquisadas e os registros recuperados. O tempo de execução do processo de inserção e de pesquisa do conjunto de dados é exibido ao final.

3.1 Estrutura dos arquivos no diretório do programa

Main.cpp: esse é o arquivo que contém o programa principal, instruções para inicialização e utilização do programa a partir de um menu construído com o switch case. Nele contém as declarações iniciais de variáveis e objetos além de chamadas de funções que irão inicializar o processo de inserção e recuperação de registros por meio da tabela hash extensível.

FileManagement.h, simulation.h, hash.h: todos esses são arquivos que contém os cabeçalhos dos membros e funções de cada classe acompanhado de um comentário que define a funcionalidade básica de determinado procedimento ou variável dentro do contexto de execução do programa.

FileManagement.cpp: dentro desse arquivo há o desenvolvimento algorítmico das funções explicitadas no fileManagement.h. Entretanto, uma menção especial merece ser dada à 5 funções que estão relacionadas intrinsecamente uma com a outra no contexto de manipulação do sistema de arquivos criadas conforme as regras de funcionamento da tabela hash extensível. O diagrama de classes UML pode ser visualizado na Figura 2.

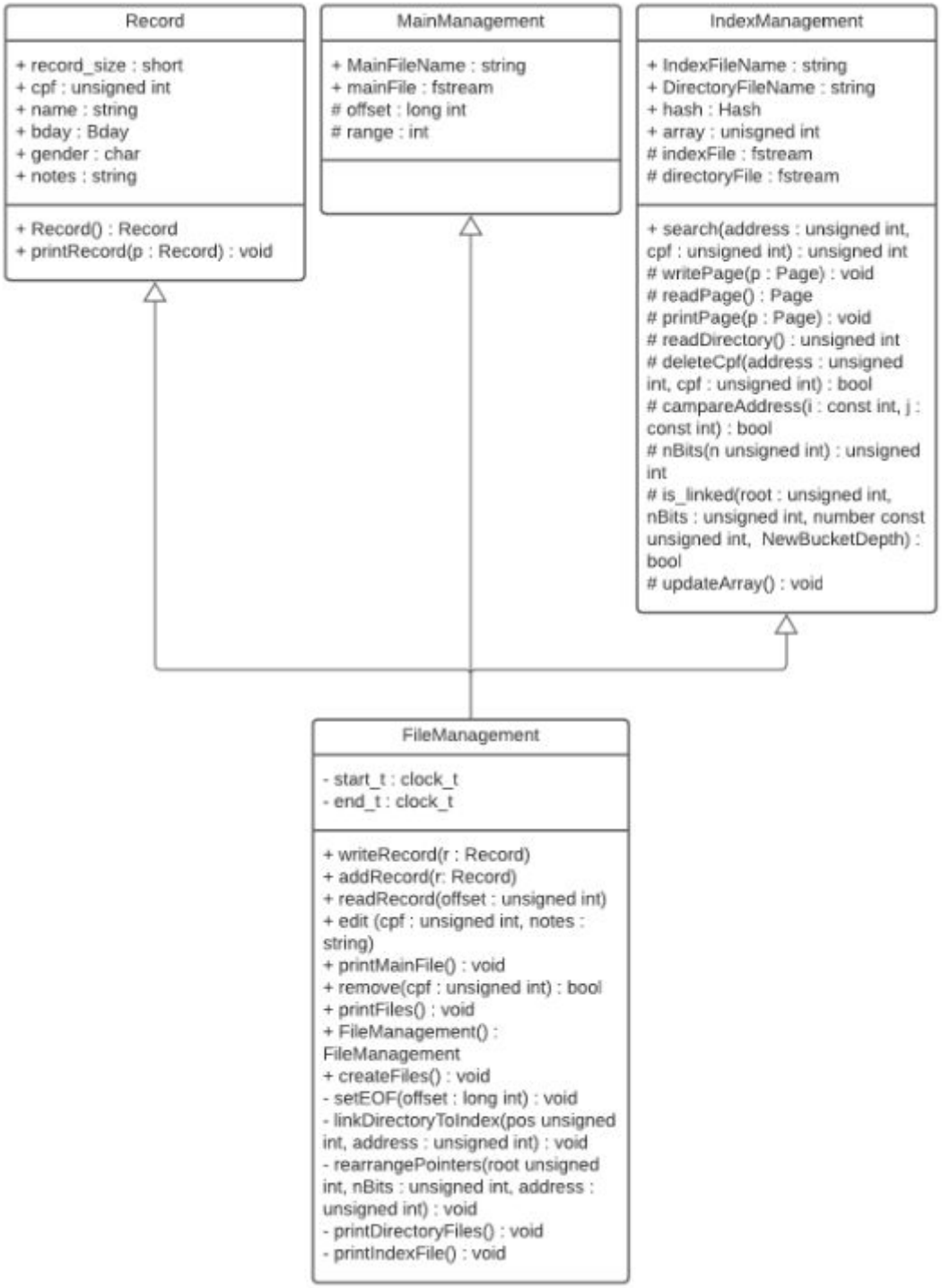


Figura 2. Diagrama de classes UML do arquivo fileManagement.cpp

São essas funções: writeRecord, LinkDirectoryToIndex, LinkDirectoryToMain, is_linked e RearrangePointers. Detalhes acerca da lógica de programação envolvida por trás das funções writeRecord, linkDirectoryToIndex e LinkDirectoryToMain não serão abordados aqui, pois já foram detalhado nos comentários do programa.

Para entender como essas funções se relacionam uma com a outra e até para compreender algumas partes comentadas no programa se faz necessário a definição de alguns termos que são usados na explicação do código. A primeira delas é o termo ponteiro que se refere a cada posição do vetor do diretório e não especificamente o aquela posição guarda, mas sim a posição em si. Um ponteiro irá guardar sempre um valor de endereço de uma página no índice, o que é o mesmo que dizer que o ponteiro referência uma página do índice. Tendo isso em vista, para uma melhor análise e entendimento de como essas funções trabalham juntas deve-se analisar algumas partes importantes do código onde essas funções se interceptam.

Nessa parte pode-se enxergar como a função linkDirectoryToIndex se relaciona diretamente com a função LinkIndexToMain (Figura 3), pois as medidas a serem tomadas por esta depende totalmente do resultado retornado pela função responsável por manipular o arquivo de índices, isto é, o LinkIndexToMain. Isso se dá pelo fato de que o bucket retornado pela função que linka o index e o main atua como "bucket flag" e ele juntamente com a flag de inserção da hash (mencionado no programa) definem as medidas tomadas pela função que manipula o diretório, medidas como por exemplo, expansão do diretório e rearranjo dos ponteiros.

```
/*
A linha de código logo a seguir chama a função responsável por armazenar
o novo bucket no índice e fazer o link entre o arquivo mestre e o índice.
Ela passa como argumento o endereço da página o bucket deve ser inserido
e o bucket que deve ser inserido.
*/

bucket = LinkIndexToMain(array[hash.hashFunction(cpf)], b);
```

Figura 3. linkDirectoryToIndex

Nesse caso percebe-se a relação da função de manipulação do diretório (LinkDirectoryToIndex) com a função que faz o rearranjo dos ponteiros. Para entender como o rearranjo de ponteiros ocorre é necessário primeiro compreender que uma das regras da hash extensível é a de que os ponteiros novos gerados após a expansão do diretório apontam naturalmente para um outro ponteiro raiz correspondente que já foi criado anteriormente em uma outra expansão. Para entender melhor, segue abaixo o antes e depois (Figura 4) respectivamente de uma expansão do diretório de uma hash extensível fictícia:

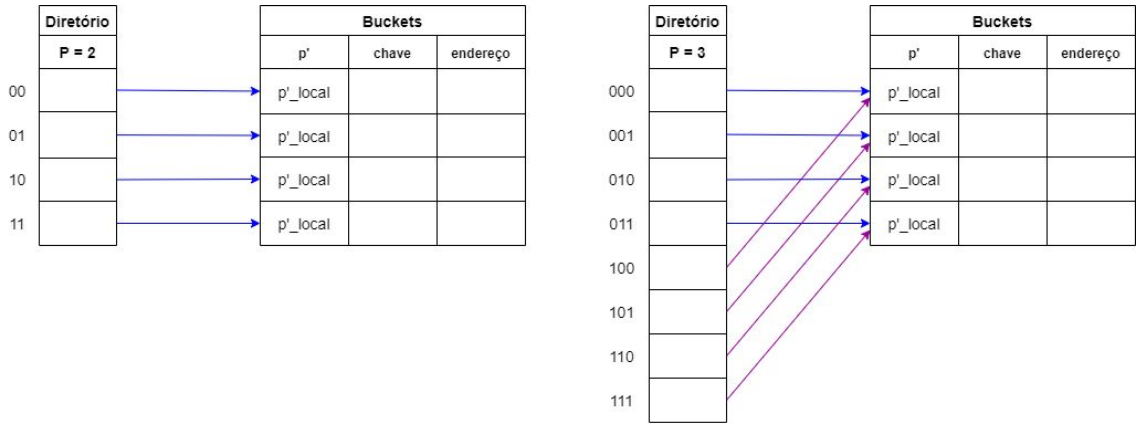


Figura 4. Antes e depois da expansão do diretório de uma hash extensível

O cálculo do rearranjo de ponteiros como visto pode seguir o seguinte modelo matemático:

$$novo\_ponteiro = novo\_ponteiro - pow(2, profundidade\_global) / 2$$

Recursivamente e, dado o ponteiro raiz inicial como argumento, a função RearrangePointers (Figura 5) gera todos os ponteiros que podem estar apontando para o ponteiro raiz desde o primeiro ponteiro até o maior o qual pode ser representado com uma quantidade de número de bits igual a profundidade global. Além disso, ao mesmo tempo, dado o número de bits do ponteiro raiz mais 1 e o endereço do índice que irá ser apontado pelo ponteiro raiz que originou a recursividade, ele compara se o endereço apontado pelos ponteiros gerados são iguais ao endereço apontado pelo ponteiro raiz que os gerou. Caso eles forem iguais, o ponteiro gerado recebe o endereço do índice.

```
/*
Caso a função retorne diferente de 1, isso quer dizer que ela retornou um
endereço de uma nova página que foi criada,mas ainda não foi possível
inserir o bucket, pois todos os elementos caíram na nova página dps do
rehash.
Sendo assim é necessário fazer o rearranjo novo dos ponteiros por conta
da nova página criada.
*/

RearrangePointers(bucket.cpf, nBits(bucket.cpf) + 1, bucket.Address);
//faz o rearranjo dos ponteiros do diretório

/*
OBS: A função linkIndexToMain atribui ao cpf nesse caso uma posição do
vetor do diretório o qual o endereço da nova página deve ser atribuído
*/

array[bucket.cpf] = bucket.Address; //atribui determinada posição do
vetor do diretório o endereço da nova página
```

Figura 5. Função RearrangePointers

Abaixo na Figura 6 segue uma tentativa de exemplificação em diagrama dos ponteiros gerado pela recursividade dado um ponteiro raiz inicial:

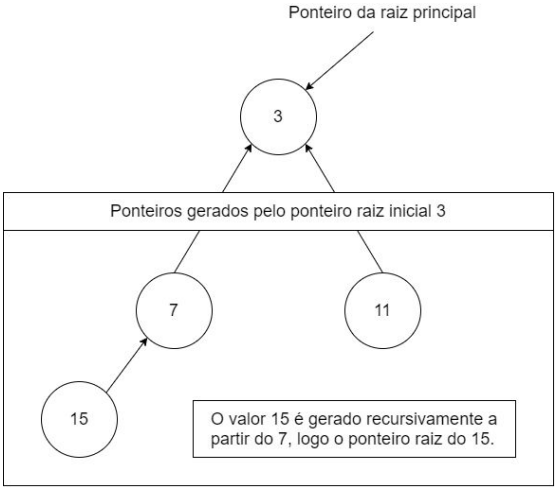


Figura 6. Diagrama dos ponteiros gerado pela recursividade

É importante notar que para exemplificação o diagrama acima foi baseado num diretório com profundidade global de 4. Abaixo segue a Figura 7 do algoritmo RearrangePointers:

```
void fileManagement::RearrangePointers(unsigned int root, unsigned int nbits,
                                       unsigned int address)
{
  if (nbits < hash.depth)
  {
     RearrangePointers(root, nbits + 1, address);
     RearrangePointers(root + (unsigned int)(pow(2, nbits) / 2), nbits + 1,
                       address);
  }
  if (nbits <= hash.depth)
  {
     unsigned int index = root + unsigned int(pow(2, nbits) / 2);
     unsigned int bitsNumber = nBits(index);

     if (compareAddresses(index,index - unsigned int(pow(2,bitsNumber) / 2)))
     {
        array[index] = address;
     }
  }
}
```

Figura 7. Algoritmo RearrangePointers

Neste último caso vemos a interação da função LinkIndexToMain com a função is_linked (Figura 8). Similarmente a função RearrangePointers, a função is_linked gera os ponteiros dado um ponteiro inicial raiz. Tanto a função RearrangePointers quanto a função is_linked gera os ponteiros da mesma forma, ou seja, as duas possuem o mesmo motor recursivo que possibilita a formação desses apontadores. Como a funcionalidade da função is_linked já foi detalhado no comentário do programa, não há necessidade desta ser reproduzida no presente relatório.

```
/*
Faz o rehash com todos os elementos conectados a raiz. A raiz é a posição no
vetor do diretório que irá referenciar a nova página a ser inserida.Os
elementos conectados/referenciados a raiz pode ser obtido da seguinte forma:
             elemento = raiz + pow(2,quantidade de bits)/2
Sendo que a quantidade de bits está no seguinte intervalo de números
inteiros:
    quantidade de bits da raiz < quantidade de bits <= profundidade global

OBS: para calcular todos os elementos é necessário que a quantidade de bits
assuma todos os valores possíveis dentro daquele intervalo.
Isso faz parte da função is linked e será detalhado por meio de um simples
diagrama como exemplo no relatório.
*/

if (is linked(key, NumberOfBits + 1, temp2.b[i].cpf, temp2.bucketDepth + 1))
{
   temp.b[temp.nBuckets] = temp2.b[i];
   temp2.nBuckets--;
   temp.nBuckets++;
}
```

Figura 8. Parte da função LinkIndexToMain com a função is_linked

Abaixo segue na Figura 9 o algoritmo da função is_linked:

```
bool indexManagement::is linked(unsigned int root, unsigned int nbits,
                                const unsigned int number,
                                const unsigned int NewBucketDepth)
{
  if (nbits < hash.depth)
  {
     return (number % (int)pow(2, NewBucketDepth) == root
            + (int)(pow(2, nbits) / 2)) || is linked(root, nbits + 1, number,
            NewBucketDepth) || is linked(root + (int)(pow(2, nbits) / 2),
            nbits + 1, number, NewBucketDepth);
  }
  else
  {
     return (number % (int)pow(2, NewBucketDepth) == root + (int)(pow(2,
            nbits) / 2));
  }
}
```

Figura 9. Função is_linked

A respeito dos arquivos hash.cpp e simulation.cpp do programa, não será realizado uma abordagem sistemática destes no presente relatório, pois, além dos comentários conseguirem cobrir a explicação deles, não há nenhuma funcionalidade complexa por trás do desenvolvimento das funções contidas nesses dois arquivos. Ambos documentos servem para integrar o arquivo principal que gerencia todo o processo da hash extensível que é o fileManagement.

Segue abaixo na Figura 10 o diagrama UML do arquivo hash.cpp.

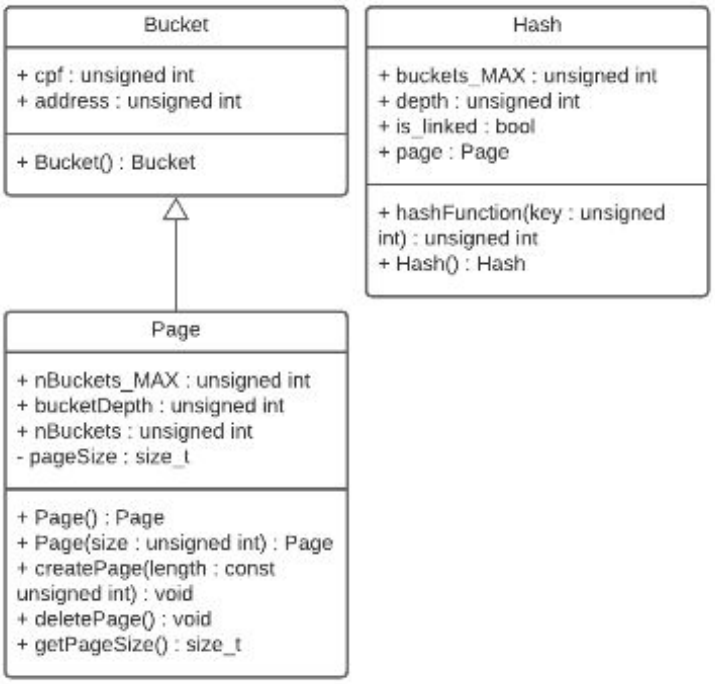


Figura 10. Diagrama de classes UML do arquivo Hash.cpp

Segue abaixo na Figura 11 o diagrama UML do arquivo simulation.cpp.

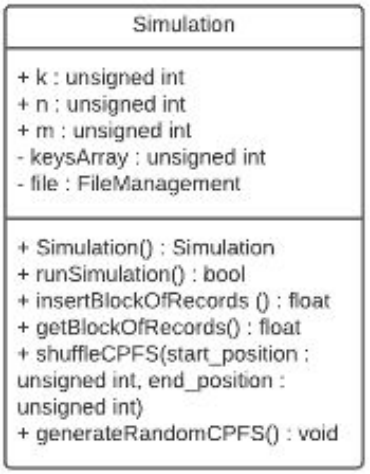


Figura 11. Diagrama de classes UML do arquivo simulation.cpp

Completando a análise do programa, segue o diagrama esquemático de como se organiza o sistema de gerenciamento de arquivo, como se pode observar na Figura 12.

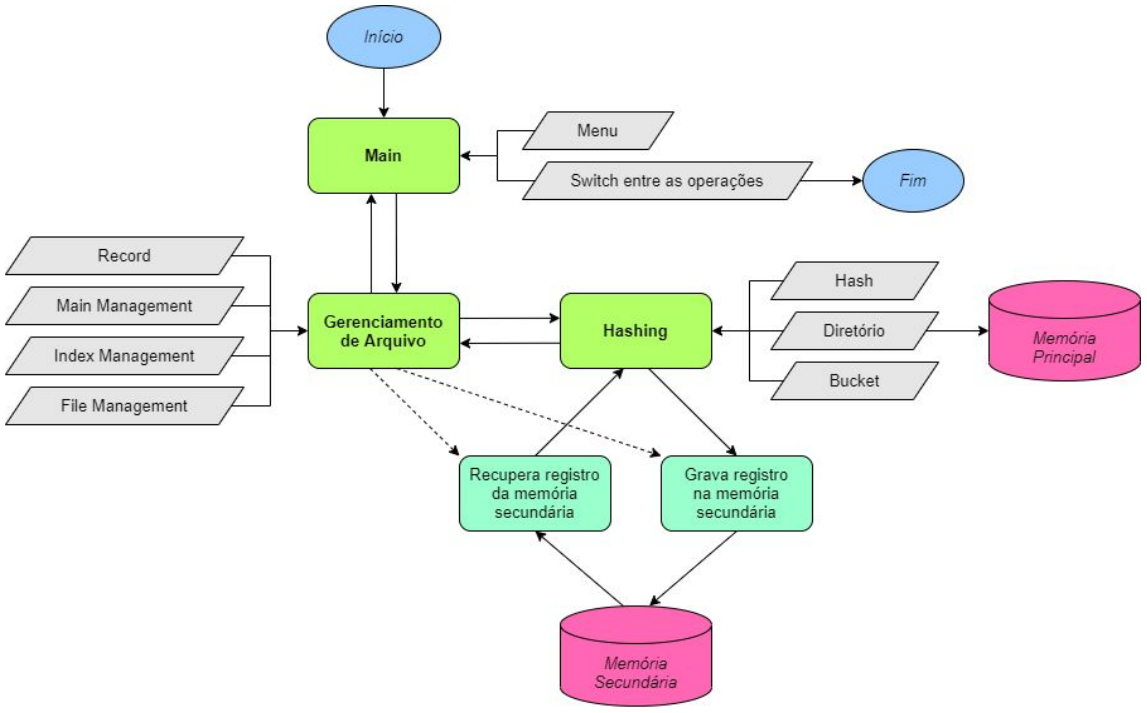


Figura 12. Fluxograma e funcionalidades do programa

3.2. Plataforma de Desenvolvimento e Linguagem Utilizada

O programa foi desenvolvido na IDE (Integrated Development Environment) Visual Studio 2019 e a linguagem adotada foi o C++ dado as suas características ao possuir recursos orientados a objetos, genéricos e funcionais, além de recursos para manipulação de memória de baixo nível e excelente desempenho, eficiência e flexibilidade de uso em sistemas.

Também é uma linguagem oferece velocidade de execução rápida e bom controle sobre a memória com a presença de uso de classes, recursos determinísticos e alocação de memória dinâmica e possibilita acesso direto ao hardware em nível de máquina.

3.3. Ambiente de Execução do Sistema

A utilização de duas máquinas possibilitou uma análise mais completa na simulação do programa. As especificações genéricas do hardware e do sistema operacional destes laptops são:

Máquina 1:

- Processador Intel Core i7-7700HQ 7rd gen CPU 2.80 GHz, cache de 6MB
- Memória RAM 16GB DDR4 2.133 MHz
- HD, SATA 3.0, capacidade de armazenamento de 1 TB
- Sistema operacional 64 bits Windows 10 Home versão 2004

As especificações do outro laptop utilizado:

Máquina 2:

- Processador Intel Core i5-3210M 3rd gen CPU 2.50 GHz, cache de 3MB
- Memória RAM de 4 GB DDR3 1.333MHz
- HD, SATA 3.0, capacidade de armazenamento de 500GB
- Sistema operacional 64 bits Windows 10 Pro versão 2004

Em ambos os casos também foi utilizado uma unidade de memória flash (EEPROM) no barramento USB 2.0 para realizar os testes de acesso e recuperação dos dados uma pendrive.

4. Análise dos Resultados

Nesta implementação o hashing extensível é o "core" para indexação das informações, associando uma chaves de pesquisa aos dados do registro e gerenciamento de forma dinâmica e eficiente o arquivos armazenados em memória secundária. Os parâmetros de tempo para se analisar o desempenho do programa foram obtidas através da função clock(), da biblioteca time.h, que permitiu medir o tempo de execução de determinadas operações do programa, com isso foi possível realizar as comparações entre as máquinas e dispositivos de armazenamento e assim entender o comportamento do programa de cada operação.

O resultado dos testes (Figura 13) advém da execução da função "Run a simulation" e o desempenho das duas máquinas ao rodar o programa foram comparados, para tal foi estabelecido uma sequência crescente de conjuntos de registros, iniciando em 1 mil, 10 mil, 100 mil e terminando a simulação com a inserção e recuperação de um conjunto de 1 milhão de registros cujo o tamanho de armazenamento alcança 1,17 GB. Como parâmetros fixos foram estabelecidos o tamanho do registro com 1200 Bytes e 4000 buckets por página.

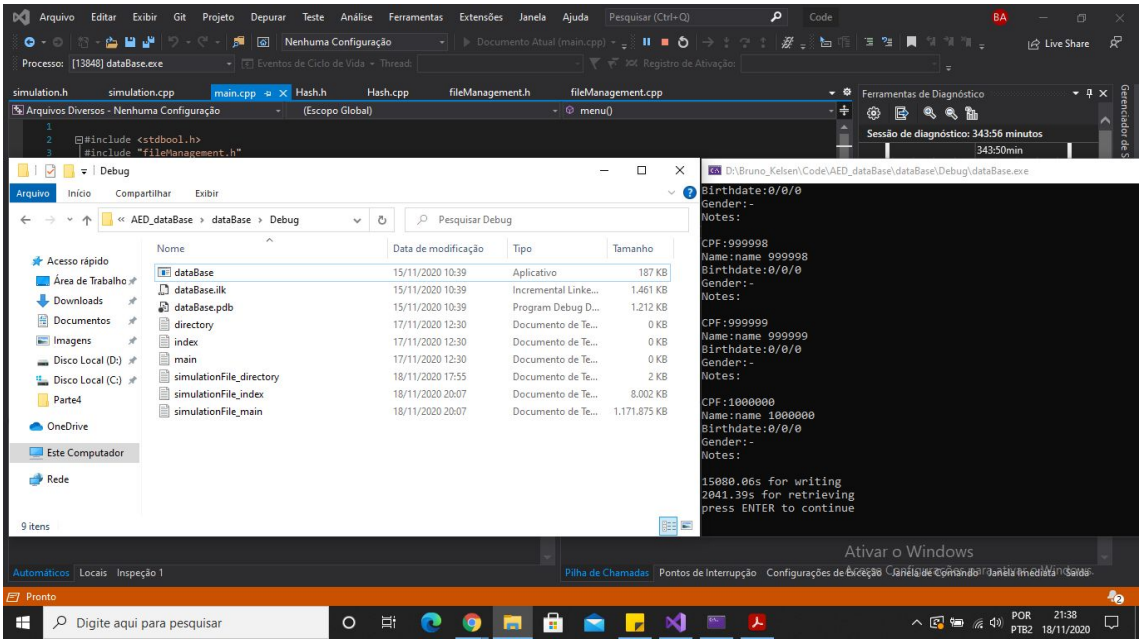


Figura 13. Execução e teste do programa

Os resultados obtidos nos testes foram tabulados, o resultado da Máquina 1 (Tabela 1), dado pela quantidade de registos "N_Files", mostra o aumento gradual em KBytes do diretório e do bucket do hashing extensível, em que "File_directory" é o diretório e o "File_index" é o bucket, sendo o "File_Main" o arquivo-mestre o tempo em segundos que se levou para escrever uma determinada quantidade de registros no arquivo-mestre é dado em "Time_writing", sendo que "Time_retrieving" foi o tempo necessário para recuperar todos os registro que posteriormente são exibidos.

Tabela 1. Teste em disco na Máquina 1

N_files	File_directory(KB)	File_index(KB)	File_main(KB)	Time_writing(s)	Time_retrieving(s)
1000	1	32	1172	3,03	3,87
10000	1	126	11719	26,47	18,82
100000	1	1001	117188	341,15	168,21
1000000	2	8002	1171875	3429,37	1701,08

Nos teste realizados em disco (HD) a Máquina 1 levou 3s para inserir e recuperar 1 mil registros, 26s para inserir e 18s para recuperar os 10 mil e 5 minutos para inserir e 2 minutos para recuperar os 100 mil, como a intenção é analisar resultados em grandes quantidades de registros essa máquina levou 57 minutos para criar um arquivo de 1,17 GB de registros e 28 minutos para recuperar todos os registros.

Na Tabela 2 os resultados dos testes realizados em um pendrive, nesse caso a Máquina 1 levou 30s para inserir e 3s para recuperar os 1 mil registros, 4 minutos para inserir e 18s para recuperar os 10 mil, mas para inserir 100 mil registros essa máquina

levou 9 horas e 35 minutos, e 2 minutos para realizar a operação de recuperação. Dado a demora de tempo ao realizar o teste de 100 mil, foi feita uma estimativa de quanto tempo essa máquina levaria para criar um arquivo de 1,17 GB e esse tempo seria de em torno de 190,4 horas, ou seja 7 dias. Ao comparar com as operações com arquivo em disco, no pendrive estas são bastante lentas para inserir grandes volumes de dados, o mesmo não acontece ao fazer a recuperação dos dados.

Isso acontece pelo fato do pendrive ser uma memória auxiliar EEPROM, que se conecta na entrada USB 2.0, e mesmo essa sendo a taxa máxima, não a real, mas que o dispositivo pode atingir, esse barramento de acesso USB possui uma velocidade de transferência em torno de 480 Mb/s, o equivalente a cerca de 60 MB por segundo.

Já a unidade interna, o HD utiliza o barramento SATA 3.0 de 6 Gb/s, que oferece uma frequência de 6 GHz e taxa de transmissão de até 600 Mb/s, além de outros fatores relacionados ao gerenciamento pelo sistema operacional e a tecnologia do hardware e sua microeletrônica. Dessa forma, o barramento USB se torna um gargalo elevando o tempo que o gerenciador exige durante o processo de inserção do arquivo no pendrive.

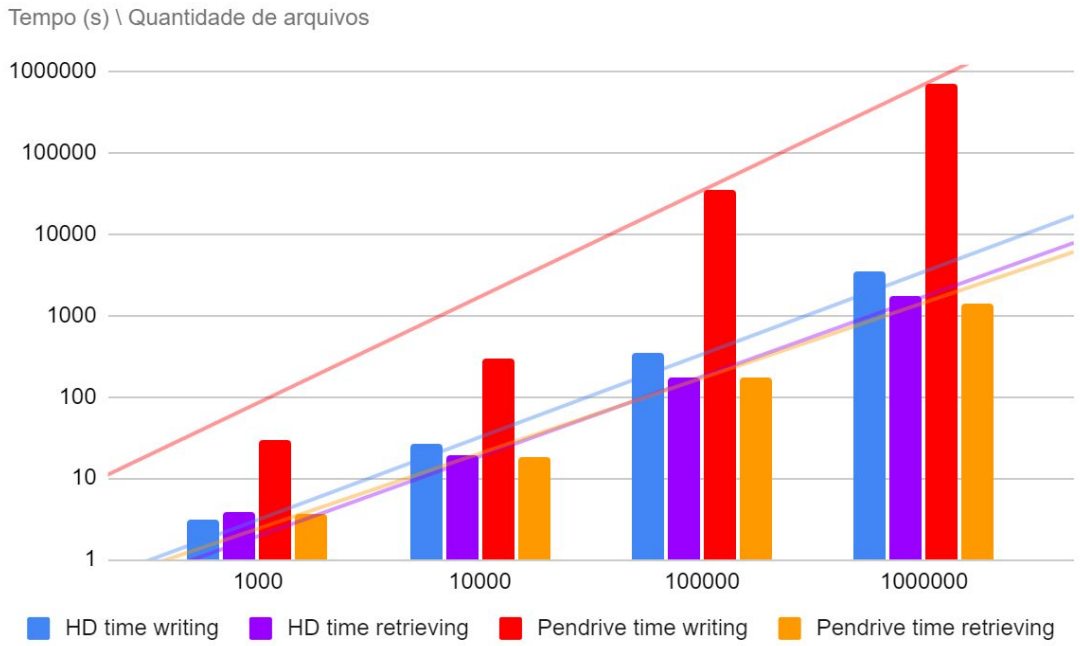
Tabela 2. Teste em pendrive na Máquina 1

N_files	File_directory(KB)	File_index(KB)	File_main(KB)	Time_writing(s)	Time_retrieving(s)
1000	1	32	1172	29,94	3,73
10000	1	126	11719	293,59	18,2
100000	1	1001	117188	35160,09	168,46
1000000	1	8002	1171875	685621,75*	1431,91*

*tempo estipulado

No Gráfico 1 é possível visualizar a relação de tempo de inserção e recuperação em disco (HD) e no pendrive na Máquina 2.

Gráfico 1. Comparativo em disco e em pendrive na Máquina 1



Nos teste (Tabela 3) realizados em disco (HD) a Máquina 2 levou 19s para inserir e 8s recuperar os 1 mil registros, 2 minutos para inserir e 20s para recuperar os 10 mil e 29 minutos para inserir e 3 minutos para recuperar os 100 mil, os resultados em grandes quantidades de registros essa máquina levou 4 horas para criar um arquivo de 1,17 GB de registros e 34 minutos para recuperar todos os registros.

Tabela 3. Teste em disco no Máquina 2

N_files	File_directory(KB)	File_index(KB)	File_main(KB)	Time_writing(s)	Time_retrieving(s)
1000	1	32	1172	19,02	8,59
10000	1	126	11719	143,7	19,97
100000	1	1001	117188	1748,33	185,33
1000000	2	8002	1171875	15080,06	2041,39

Sendo que abaixo na Tabela 4 estão os resultados dos testes realizados no pendrive.

Tabela 4. Teste em pendrive na Máquina 2

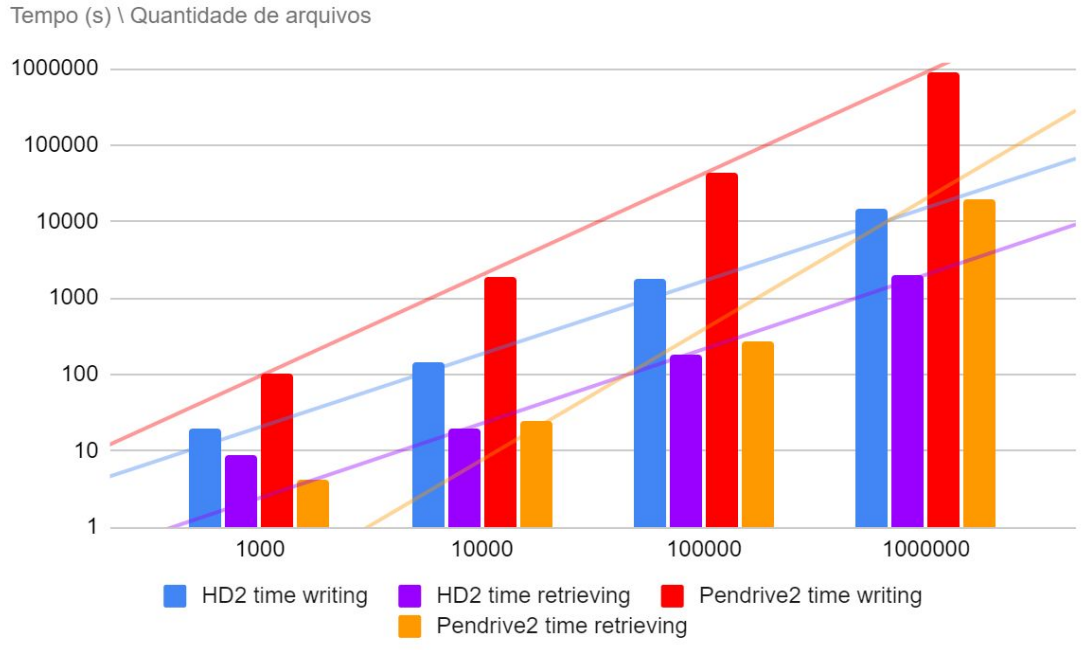
N_files	File_directory(KB)	File_index(KB)	File_main(KB)	Time_writing(s)	Time_retrieving(s)
1000	1	32	1172	101,79	4,07
10000	1	126	11719	1840,15	24,7
100000	1	1001	117188	42814,43	270,72
1000000	2	8002	1171875	899103,03*	20013,23*

*tempo estipulado

Neste caso a Máquina 2 levou 2 minutos para inserir e 4s para recuperar os 1 mil registros, 30 minutos para inserir e 24s para recuperar os 10 mil, mas para inserir 100 mil registros essa máquina levou respectivamente 12 horas e 4 minutos, e mais 4 minutos para realizar a recuperação dos registos. Como as operações de 1.17 GB levam tempo relativamente longa para serem concluídos, também foi feita uma estimativa de quanto tempo essa máquina levaria, esse tempo seria de em torno de 249,7 horas, ou seja 10 dias.

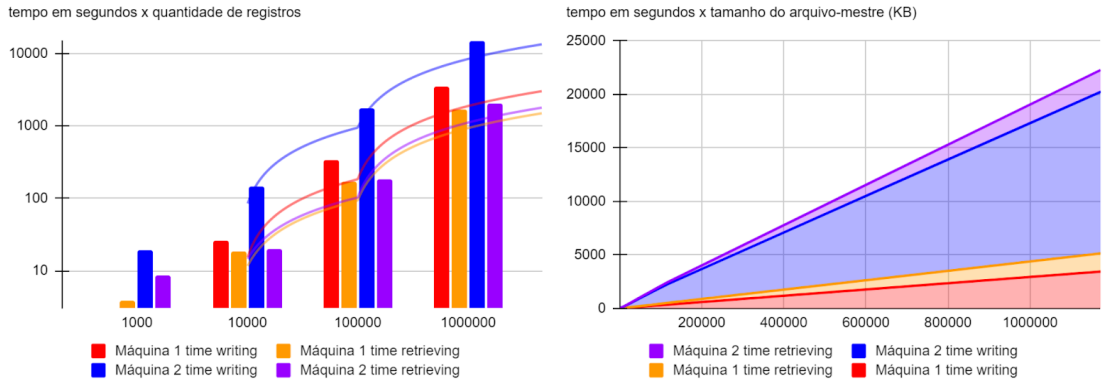
No Gráfico 2 é possível visualizar a relação de tempo de inserção e recuperação em disco (HD) e no pendrive na Máquina 2.

Gráfico 2. Comparativo em disco e em pendrive na Máquina 2



Um das propostas alcançadas nessa análise foi comparar o tempo de execução entre as duas máquinas (Gráfico 3), o intuito aqui é avaliar o quanto as diferenças de hardware influenciam no desempenho do programa.

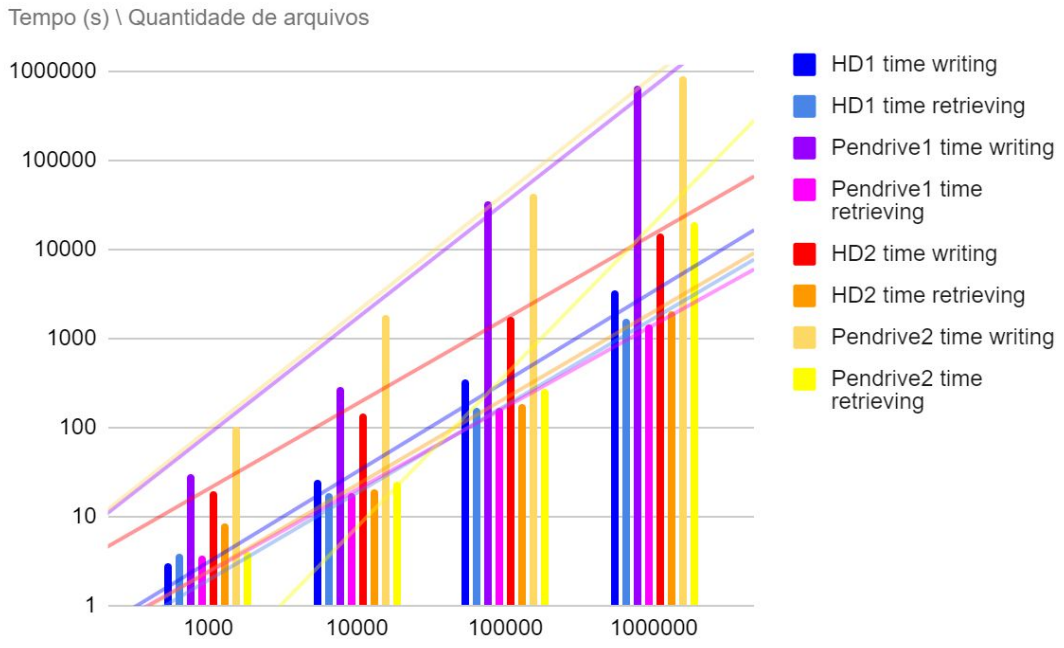
Gráfico 3. Comparativo entre as duas máquinas



Os gráficos acima mostram duas situações um relação do tempo pela quantidade de registros inseridos e outro de tempo pelo tamanho do arquivo armazenado em KiloBytes.

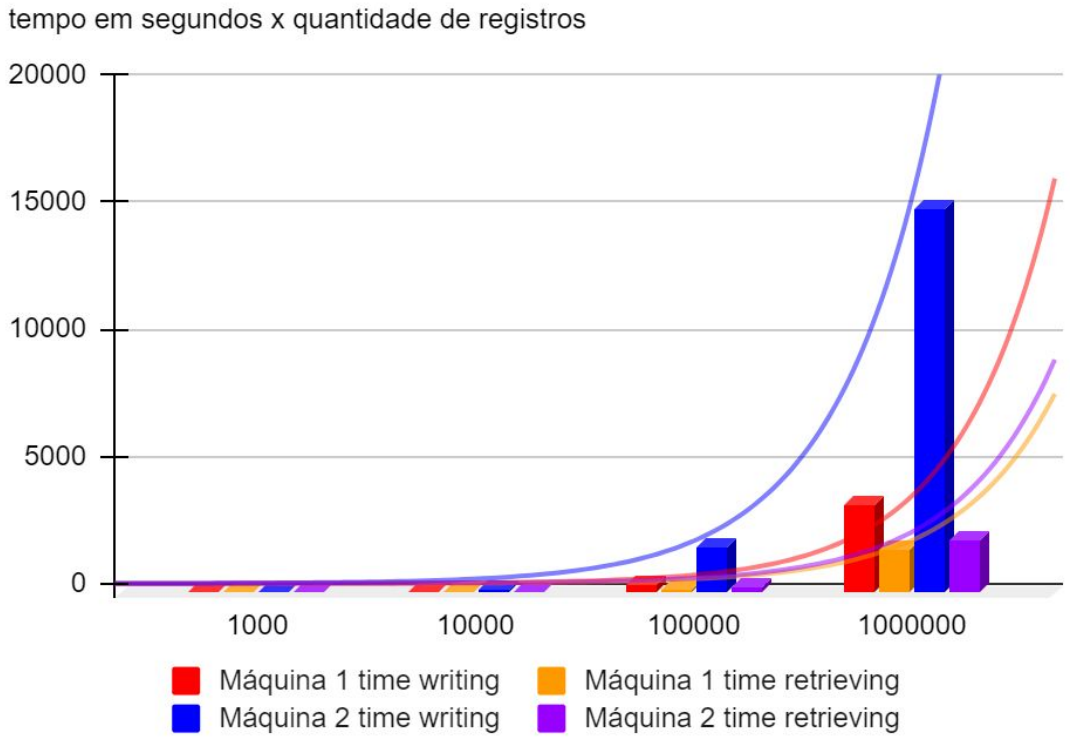
No Gráfico 4 é possível visualizar a relação de tempo de inserção e recuperação em disco (HD) e no pendrive em ambas as máquinas, onde sufixo 1 se refere a Máquina 1 (de melhor performance) e 2 a outra máquina (de menor performance).

Gráfico 4. Comparativo entre as duas máquinas nas operações no HD e no pendrive



As diferenças das máquinas ficam perceptíveis ao comparar o tempo de escrita em grandes números de registros e de recuperação dos mesmos (Gráfico 5), a Máquina 1 gastou inicialmente 3s, 26s, 5 minutos e 57 minutos no teste em arquivo de 1.17 GB quanto a Máquina 2 gastou sequencialmente 19s, 2 minutos, 29 minutos e 4 hora para realizar os testes com o arquivo de 1.17 GB, sendo assim 3 vezes mais lenta no processamento do que a Máquina 1. Portanto, confirmando que as características do hardware irão influenciar no desempenho do programa uma vez que essa é a principal diferença, já que ambas as máquinas usam o mesmo sistema operacional.

Gráfico 5. Comparativo de inserção e recuperação em grandes números de registros



Por fim os resultados obtidos nesta implementação foram importantes e optar por testar o sistema de gerenciamento em duas máquinas e comparar o desempenho do programa foi o diferencial aqui proposto. Sendo que isso revelou informações interessantes de como que as configurações de cada máquina influenciou na execução dos testes.

3. Considerações Finais

Um dos maiores desafios da computação atualmente é lidar com problemáticas relacionadas ao tempo de execução dos algoritmos de manipulação de dados. Ao longo da história da tecnologia da informação, foram propostas muitas soluções de otimização

com foco nesse tema. Este trabalho trouxe a construção, operação e análise de uma dessas técnicas, o hashing extensível.

Em relação à escrita do algoritmo, foram detalhadas as informações sobre a linguagem, sobre a IDE utilizada, sobre os arquivos elaborados no projeto e foi exibido o diagrama UML das classes construídas. A respeito da execução, foram utilizadas unidades flash de memória e dois computadores portáteis com processadores diferentes para coleta de dados sobre o desempenho. Finalmente, os dados coletados foram comparados, sendo úteis para a percepção de que o hardware teve grande impacto na eficiência de cada um dos eventos. Sendo que em todos os casos o tempo de escrita foi mais extenso do que o tempo de recuperação nas duas máquinas e nos dois pendrives. Analisando os gráficos, percebeu-se que lidando com a escrita de uma quantidade de 1 milhão de registros, a máquina de processador i5, teve um tempo final de execução aproximadamente quatro vezes maior do que o mesmo procedimento realizado na máquina de processador i7. Já o teste realizado em pendrive se mostrou aproximadamente três vezes mais rápido no laptop i7 com mil registros e aproximadamente treze vezes mais rápido com um milhão de registros na mesma máquina. Ambos computadores mostraram uma velocidade de escrita na memória flash mais lenta do que em relação à escrita no disco rígido.

O trabalho realizado foi muito importante em termos de melhor compreensão de conceitos, das técnicas e da implementação prática de uma estrutura de manipulação de dados muito utilizada em grandes sistemas devido a seu alto desempenho. Com a conclusão do desafio proposto, foi possível elucidar alguns dos temas abordados na disciplina de Algoritmos e Estrutura de Dados III. Destes destaca-se o comportamento real da execução de técnicas de indexação dinâmica de arquivos, detalhes da lógica de montagem do código e também as análises de eficácia do sistema aqui desenvolvido, sendo possível reconhecer quais pontos foram importantes no desenvolvimento deste trabalho e outros que podem ser otimizados.

Referências

Fagin, R. & Nievergelt, J. & Pippenger, N. & Strong, H.. (1979). Extendible Hashing - A Fast Access Method for Dynamic Files.. In: ACM Trans. Database Syst.. v. 4. 315-344. 10.1145/320083.320092.