Algoritmo de ordenação externa baseado em Merge Sort

Jairo F. Gez

Universidade Federal de Santa Maria(UFSM)
Santa Maria – RS – Brasil

jfgez@inf.ufsm.br

Abstract. *This article describes the code developed for the implementation of an external ordering algorithm based on Merge Sort and the techniques involved in its process.*

Resumo. *Este artigo descreve o código desenvolvido para a implementação de um algoritmo de ordenação externa baseado em Merge Sort e as técnicas envolvidas em seu processo.*

1. Introdução

Ordenar registros de um arquivo muito grande pode se tornar um desafio quando eles têm tamanho maior que a memória interna disponível, uma solução para esse problema é dividir o arquivo em menores partes, guardando-as ordenadamente na memória secundária e em seguida uni-las novamente. Chamamos a primeira parte deste processo de "classificação" e a segunda de "intercalação". Alguns algoritmos que usam essa ideia são baseados em Quick Sort e outros em Merge Sort. O algoritmo desenvolvido no presente trabalho foi baseado em Merge Sort.

2. Algoritmo

A primeira parte do algoritmo, desenvolvido em C#, divide o arquivo em partes menores que não ocupem mais de 10KB(valor arbitrário), esse tamanho foi escolhido para dividir o arquivo, em média, em 10 partes, uma vez que os arquivos de exemplo

```
static void Divide(string arquivo)
{
    int numParte = 1;
    StreamWriter sw = new StreamWriter(string.Format(".\\parte{0}.txt", numParte));
    using (StreamReader sr = new StreamReader(arquivo))
    {
        while (sr.Peek() >= 0)
        {
            /* Escreve o registro */
            sw.WriteLine(sr.ReadLine());

            /* Se chegou ao limite de tamanho que impusemos, fecha e cria nova parte */
            /* Ou se era a última linha simplesmente fecha */
            if (sw.BaseStream.Length > 10000 && sr.Peek() >= 0)
            {
                sw.Close();
                numParte++;
                sw = new StreamWriter(string.Format(".\\parte{0}.txt", numParte));
            }
        }
    }
    sw.Close();
}
```

Figura 1: Dividindo o arquivo em partes não maiores que 10KB

disponibilizados no site da disciplina tem em média 100KB. O arquivo deve estar no mesmo diretório do executável e estar nomeado como ‘Arquivo.txt’.

Feita a divisão, cada pedaço do arquivo é ordenado, via Merge Sort. São criados novos arquivos ordenados referentes a cada pedaço menor e os arquivos antigos desordenados são deletados. A esta fase chamamos de classificação.

```
static void Classifica()
{
    foreach (string pedaco in Directory.GetFiles(".\\", "parte*.txt"))
    {
        /* coloca todos os registros dos pequenos arquivos, um por vez, dentro de um array e os ordena via mergeSort */
        string[] registros = File.ReadAllLines(pedaco);
        MergeSort(registros);
        /* cria o pedaço ordenado e deleta o antigo */
        string parteOrdenada = pedaco.Replace("parte", "parteOrdenada");
        File.WriteAllLines(parteOrdenada, registros);
        File.Delete(pedaco);
    }
```

Figura 2: Fase de classificação

O algoritmo Merge Sort(consta no projeto) não será mostrado no artigo por ser de simples implementação e já ter sido visto em aula.

Depois vem a intercalação, a parte realmente complicada de implementar: o método criado abre uma FIFO com todos os trechos classificados simultaneamente (K-way merge). Em seguida, seleciona o registro com o menor valor entre as filas classificadas e escreve num novo arquivo final chamado “ArquivoOrdenado.txt”. O tamanho escolhido para cada fila foi 10 registros e quando cada fila é esvaziada o algoritmo verifica se o arquivo ainda tem registros disponíveis antes de fazer a próxima iteração. Havendo registros, ele preenche a fila, senão simplesmente faz ela apontar para null.

```
/* Faz a intercalação entre os pedaços menores */
static void Intercala()
{
    string[] partes = Directory.GetFiles(".\\", "parteOrdenada*.txt");
    int qtdPedacos = partes.Length; /* número de pedaços em que o arquivo foi dividido */
    int tamanhoBuffer = 10; /* número de registros a serem guardados em cada fila(buffer) */

    /* abre os arquivos usando um streamReader que carrega um ponteiro para o inicio de cada arquivo */
    StreamReader[] readers = new StreamReader[qtdPedacos];
    for (int i = 0; i < qtdPedacos; i++)
        readers[i] = new StreamReader(partes[i]);

    /* monta as filas */
    Queue<string>[] filas = new Queue<string>[qtdPedacos];
    for (int i = 0; i < qtdPedacos; i++)
        filas[i] = new Queue<string>(tamanhoBuffer);

    /* carrega as filas */
    for (int i = 0; i < qtdPedacos; i++)
        CarregaFila(filas[i], readers[i], tamanhoBuffer);

    /* faz a intercalação propriamente dita */
    StreamWriter sw = new StreamWriter(".\\ArquivoOrdenado.txt");
    bool terminou = false;
    int menorIndice, j;
    string menorValor;
```

Figura 3: Primeira parte do método de intercalação

Note que na primeira parte do método de intercalação são criadas instâncias da classe StreamReader(nada mais que ponteiros para o trecho de arquivo que está sendo lido) para os arquivos parcialmente ordenados, cria e preenche as filas e cria um novo arquivo final onde estarão os registros totalmente ordenados.

```
    while (!terminou)
    {
        /* encontra o pedaço com o menor valor */
        menorIndice = -1;
        menorValor = "";
        for (j = 0; j < qtdPedacos; j++)
        {
            if (filas[j] != null)
            {
                if (menorIndice < 0 || String.CompareOrdinal(filas[j].Peek(), menorValor) < 0)
                {
                    menorIndice = j;
                    menorValor = filas[j].Peek();
                }
            }
        }
        /* encerra se todas as filas estão vazias */
        if (menorIndice == -1) { terminou = true; break; }

        /* escreve o menor valor e o remove da fila */
        sw.WriteLine(menorValor);
        filas[menorIndice].Dequeue();

        /* se a fila ficou vazia, preenche, fazendo-a apontar para null quando já não houver registros para ler no seu respectivo arquivo */
        if (filas[menorIndice].Count == 0)
        {
            CarregaFila(filas[menorIndice], readers[menorIndice], tamanhoBuffer);
            if (filas[menorIndice].Count == 0)
            {
                filas[menorIndice] = null;
            }
        }
    }
    sw.Close();

    /* fecha as partes(arquivos) e as deleta */
    for (int i = 0; i < qtdPedacos; i++)
    {
        readers[i].Close();
        File.Delete(partes[i]);
    }
}/* static void Intercala() */
```

Figura 4: Segunda parte do método de intercalação

4. Aplicações

Algoritmos de ordenação externa tem suma importância, por exemplo, em bancos de dados, desde consultas que exigem resultados ordenados(cláusula order by em SQL), até a necessidade de se ter dados ordenados para utilização de algorimos usados em operações de junções e agrupamentos.

Os sistemas de gerenciamento de banco de dados normalmente utilizam esses algoritmos para tabelas que não cabem na memória, visando diminuir o custo em termos de seeks e transferências de blocos de dados para a memória. É o caso de bancos de dados de redes sociais como, por exemplo, o FaceBook, o Instragram e o Twitter.

Referências

Ziviane, Projeto de algoritmos com implementações em Pascal e C, http://www2.dcc.ufmg.br/livros/algoritmos/slides.php

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest (2009) “Introduction to Algorithms”

Richard Cole (2018) “Parallel Merge Sort”

John Sharp (2013), Microsoft Visual C# passo a passo. Editora Intermediário.