Marden Silveira Neubert

Algoritmos Distribuídos para a Construção de Arquivos Invertidos

Dissertação apresentada ao Curso de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte

30 de março de 2000

UNIVERSIDADE FEDERAL DE MINAS GERAIS

FOLHA DE APROVAÇÃO

Algoritmos Distribuídos para a Construção de Arquivos Invertidos

MARDEN SILVEIRA NEUBERT

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

Prof. NIVIO ZIVIANI – Orientador
Departamento de Ciência da Computação – ICEx – UFMG

Prof. JOÃO PAULO FUMIO WHITAKER KITAJIMA
Instituto de Computação – UNICAMP

Prof. BERTHIER RIBEIRO DE ARAÚJO NETO
Departamento de Ciência da Computação – ICEx – UFMG

Dr. WAGNER MEIRA JÚNIOR
Departamento de Ciência da Computação – ICEx – UFMG

Belo Horizonte, 30 de março de 2000.

Resumo

Este trabalho apresenta uma família de algoritmos distribuídos visando a construção de arquivos invertidos globais para grandes volumes de texto. Dada uma coleção de documentos textuais distribuída entre várias estações de trabalho, um arquivo invertido global é um índice que permite a busca eficiente de informações no texto distribuído como um todo. Esse índice é composto por duas partes: o vocabulário global, isto é, o conjunto de palavras distintas presentes no texto distribuído, e as listas de ocorrências globais, que apontam para os documentos da coleção nos quais cada palavra do vocabulário ocorre. O ambiente de operação considerado é uma rede de alta velocidade, pela qual as estações de trabalho possam se comunicar com pouca ou nenhuma contenção. As análises supõem que o texto encontra-se distribuído igualmente entre as estações e que o índice invertido a ser gerado é consideravelmente maior que a quantidade de memória principal disponível no ambiente. As listas de ocorrências são ordenadas pela freqüência dos termos nos documentos e comprimidas de forma a reduzir as demandas de espaço e o volume de dados transferidos pelos discos e pela rede. Três algoritmos distribuídos são discutidos e seus resultados analíticos e experimentais são comparados. Os experimentos mostram que, utilizando-se quatro estações de trabalho, o mais eficiente desses algoritmos é capaz de indexar 3 gigabytes de texto em menos de 14 minutos. As análises indicam que, no mesmo ambiente de experimentação, é possível indexar uma coleção de 100 gigabytes em menos de 6 horas.

Abstract

This work presents a family of distributed algorithms to build global inverted files over large text collections. Given a document collection which is distributed among workstations in a network, a global inverted file is an index that allows fast searching in the distributed text as a whole. The index is composed by two parts: the global vocabulary – the set of all distinct words appearing in the distributed text – and the global lists of occurrences pointing to the documents in which each term in the vocabulary occurs. The operating environment considered is a high-bandwidth network of workstations which allows the machines to communicate with hardly no contention. The analysis assume that the text is evenly distributed among the workstations and that the index to be generated is considerably larger than the main memory available in the environment. The inverted lists are sorted by the frequencies of the terms in the documents and compressed in order to reduce the space requirements and the volume of data transferred through the disks and the network. Three alternatives are discussed and their analytical and experimental results are compared. The experiments show that with four machines the most efficient algorithm can invert 3 gigabytes of text in less than 14 minutes and the analysis point that in the same environment it is feasible to invert a 100-gigabyte collection in less than 6 hours.

Prefácio

Em primeiro lugar, agradeço a meus pais, Maria Aparecida Araújo Silveira e Richard Pedro Neubert, e também a minha avó, Maria Araújo Silveira, por todo o estímulo, paciência e apoio nesta jornada.

Agradeço ao professor Nivio Ziviani e também ao professor Berthier Ribeiro-Neto pela proposição deste desafio, pela confiança em mim para resolvê-lo e pela orientação ao longo de todo o trabalho. Aos demais membros da banca, professor João Paulo F.W. Kitajima e Wagner Meira Jr., agradeço pelos valiosos comentários e pela paciência em examinar todo o texto.

Agradecimentos especiais ao Victor Fernando Ribeiro e ao Miner Technology Group, hoje Brasil Online, pelo suporte financeiro ao longo de todo este trabalho, pela comprensão quando minha dedicação a ele interferia nas minhas obrigações na empresa e todo o apoio nos momentos difíceis. Agradeço também pelo estímulo nos últimos instantes desta caminhada, liberando-me da maior parte das minhas atribuições, e pela concessão das máquinas e de toda a infra-estrutura para os testes e experimentos.

Agradeço também aos amigos do Latin, o Laboratório para Tratamento da Informação, em especial ao Pável Pereira Calado e ao Edleno Silva de Moura, por todos os comentários, críticas, dicas e pelo companheirismo. Aos amigos do Projeto SIAM, Sistemas de Informação em Ambientes Móveis, em especial ao Altigran Soares da Silva, Rodrigo Barra de Almeida, Paulo Braz Golgher e Eveline Alonso Veloso, agradeço pela amizade e interesse e parabenizo-os pelo grande trabalho e competência.

Um forte abraço aos amigos *de la Revolución*, sobretudo Luciano Pereira Gomes, Hilton "Sapujo" Bruno de Almeida Sousa e Cleyton de Almeida Ferreira, pela amizade, descontração e apoio. Aos amigos e amigas da Quinta-Hard por terem me ajudado a aliviar, toda quinta-feira, as tensões e responsabilidades da semana. Um agradecimento também aos amigos do Brasil Online aqui em São Paulo, em especial ao Yuri Gitahy de Oliveira, pelo interesse e ajuda no meu trabalho.

À Karine Gomes Chaves, um agradecimento especial por todo o carinho e compreensão, pela paciência em ler e revisar todas as versões do texto e pelo estímulo nos momentos mais difíceis.

Agradeço finalmente a meu Deus, fonte de toda luz e inspiração para esta caminhada, sem a bênção do qual, certamente, nada seria feito.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

1.1 Motivação

O aumento no volume de dados armazenados na forma de textos tem acompanhado o rápido crescimento das bibliotecas digitais modernas [25], como coleções médicas e jurídicas, enciclopédias e a *World Wide Web*. Para se recuperar de forma eficiente a informação armazenada em grandes bases de dados textuais, a única solução prática é construir estruturas auxiliares, chamadas **índices**, de forma a acelerar o processo de busca. Várias técnicas especializadas de indexação foram propostas na literatura. Entre elas encontram-se as árvores PATRICIA [54], arranjos de sufixos [48], arranjos PAT [30], arquivos de assinatura [22] e arquivos invertidos [35, 85, 6], cada uma apresentando pontos fortes e fracos.

Entre tantas técnicas, porém, os **arquivos invertidos** têm sido tradicionalmente a estrutura de indexação mais utilizada. Essa popularidade deve-se ao seu custo de construção (proporcional ao tamanho do texto) e à simplicidade de sua estrutura. Um arquivo invertido é composto por duas partes: um **vocabulário**, contendo o conjunto de palavras distintas do texto, e, para cada entrada do vocabulário, uma **lista de ocorrências**, indicando os documentos da coleção nos quais aquela palavra ocorre. Cada elemento dessa lista pode incluir um peso adicional, por exemplo, a freqüência do termo no documento.

A técnica de busca utilizada por arquivos invertidos é bastante rápida, pois baseia-se no vocabulário, que é pequeno em relação ao texto. Além disso, o número de elementos do vocabulário cresce sublinearmente com tamanho do texto [37], o que garante que, mesmo para grandes coleções, o espaço demandado por ele é reduzido. Dessa forma, o vocabulário pode geralmente ser acomodado em memória principal, possibilitando pesquisas em tempo logarítmico através de busca binária ou em tempo constante quando se emprega *hashing* perfeito [20, 47]. Arquivos invertidos também oferecem suporte a diversas técnicas de recuperação de informação, como os modelos lógicos e ordenados para o processamento de pesquisas em documentos [67], técnicas de expansão de consultas [40, 88] e realimentação de relevantes [34].

Entretanto, apesar de sua estrutura simples, a construção de arquivos invertidos pode não

ser uma tarefa trivial. Quando se consideram grandes coleções textuais, as listas de documentos podem ser muito longas e exceder a memória principal disponível. Torna-se necessário, portanto, o uso de memória auxiliar em disco durante o processo de indexação, o que aumenta a complexidade do projeto de algoritmos eficientes [51, 85]. O espaço final ocupado pelo índice também pode ser bastante significativo (até 100% do tamanho do texto original), o que limita seu emprego em situações em que a área disponível para armazenamento de texto e índice é restrita.

O emprego de compressão pode reduzir significativamente o tamanho de arquivos invertidos. A técnica mais comum é dispor os números dos documentos de cada lista em ordem crescente e substituir cada um pela diferença entre ele e seu antecessor, na forma de intervalos. Como resultado, a seqüência de números de documentos é transformada em uma seqüência de valores menores, que podem ser eficientemente codificados através de métodos como Golomb [29] e Elias [21]. Para coleções típicas, o espaço ocupado pelo índice invertido comprimido representa entre 6 e 10% do tamanho do texto original [51, 85].

Uma vez construído um índice sobre uma coleção de documentos, pode-se empregá-lo para a busca da informação solicitada pelo usuário, o que, em geral, é feito da forma descrita a seguir. Primeiramente, o usuário expressa sua necessidade de informação através de uma **consulta**, composta de termos ou palavras-chave. O sistema de recuperação localiza os termos da consulta no vocabulário e obtém as listas de ocorrência correspondentes. Então, através de um **modelo** de recuperação de informação, o sistema manipula as listas na tentativa de predizer quais documentos são relevantes para o usuário. Por fim, os documentos selecionados são apresentados em ordem decrescente de relevância.

Existem vários modelos para recuperação de informação, cada um com vantagens e desvantagens; em [6, Capítulo 2] apresenta-se uma taxonomia. Dentre os modelos clássicos, destaca-se o **modelo vetorial**, que procura representar documentos e consultas através de vetores em um espaço onde as coordenadas são os termos da coleção. As freqüências dos termos nos documentos são utilizadas como pesos para favorecer documentos que contenham mais ocorrências dos termos das consultas; maiores detalhes são apresentados na Seção 2.3. Apesar de alguns pontos fracos e incorreções em sua base teórica [86], o modelo vetorial permanece como o mais utilizado em diversos tipos de sistemas de informação devido a sua eficácia, mesmo quando o usuário pouco conhece a coleção a ser pesquisada ou esta é muito grande e heterogênea, como é o caso da *WWW*.

A organização tradicional de arquivos invertidos, como encontrado na maioria das referências e algoritmos relacionados, traz os elementos das listas invertidas ordenados pelos identificadores dos documentos. Isso impõe uma dificuldade ao processamento eficiente de consultas através do modelo vetorial, pois obriga as listas a serem percorridas por completo, mesmo que a maioria de seus elementos não contribuam significativamente na ordenação final da resposta. Dessa forma, uma organização alternativa para as listas invertidas, na qual os elementos são

ordenados pelo seu peso, no caso, a freqüência do termo no documento [57, 58], permite um processamento mais eficiente, evitando que as partes finais das listas sejam lidas quando estas não são capazes de alterar a ordenação das respostas.

À medida que as coleções de documentos crescem, torna-se mais difícil gerenciar os sistemas de informação. Os custos de armazenamento, indexação e busca aumentam com o volume do texto, levando a maiores tempos de resposta e até inviabilizando o uso do sistema. Um exemplo é o caso das máquinas de busca na *Web*: seu sucesso comercial depende tanto da qualidade de suas respostas quanto de sua capacidade de processar consultas rapidamente. Como a maior parte de sua receita vem da venda de impressões de *banners* (mensagens publicitárias) em suas páginas, quanto mais consultas atendidas, maior o lucro obtido.

Uma alternativa para atender as demandas dos sistemas de informação modernos é o uso de *hardware* paralelo ou distribuído. É cada vez mais comum o uso de redes de estações, um paradigma que tornou-se conhecido como *NOW*, de *Network Of Workstations* [1]. O desempenho de estações de trabalho conectadas por uma rede rápida é comparável ao de máquinas paralelas típicas, com a vantagem de um melhor custo/benefício. Tendo em vista essa arquitetura, surgem novos algoritmos e alternativas de projeto, dos quais o presente trabalho é um exemplo.

Em [62] é proposta e analisada uma família de algoritmos distribuídos para a geração de arquivos invertidos sobre grandes bases de dados de texto. Os algoritmos são baseados na abordagem seqüencial citada anteriormente [51, 85], adaptada para gerar listas invertidas ordenadas por freqüência [58]. Supõe-se que a coleção textual encontra-se particionada entre diversas estações, conectadas por uma rede local rápida, e que a memória principal disponível nas máquinas não é suficiente para acomodar todo o arquivo invertido. O índice gerado é dito **global** [7, 63], visto que as listas contém ocorrências de termos em documentos presentes em qualquer uma das partes da coleção distribuída pela rede. Ao final do processo, cada máquina contém as listas de ocorrências de um subconjunto dos termos do vocabulário global.

Neste trabalho apresenta-se uma análise mais detalhada dos algoritmos propostos em [62], bem como do algoritmo seqüencial utilizado como base para eles. Descreve-se também o processo de implementação dessa família de algoritmos, os resultados experimentais obtidos, sua comparação com os resultados analíticos e, finalmente, apresentam-se as conclusões e propõem-se trabalhos futuros.

1.2 Trabalhos Relacionados

A técnica para construção seqüencial eficiente de arquivos invertidos comprimidos utilizando memória auxiliar em disco é apresentada em [51]. Entre as características dessa técnica, destacam-se a intercalação por vários caminhos (*multi-way merge*), que permite a ordenação do arquivo invertido lendo-o apenas uma vez, e a geração *in-situ*, pela qual o arquivo final é construído sem gastos em espaço adicional. São também revistos os códigos unário, Elias-γ,

Elias-δ e Golomb, para compressão das listas invertidas. Nesse mesmo artigo são descritos outros métodos mais simples para geração de índices, como a inversão baseada somente em memória e a ordenação em disco, que requer mais passos para ordenar o arquivo invertido e 100% de espaço temporário extra durante a construção. A adaptação desse algoritmo para o presente trabalho, os códigos utilizados e outras questões são discutidos na Seção 2.1.

A geração de funções *hashing* perfeitas empregada neste trabalho baseia-se na técnica apresentada inicialmente em [20]. Essas funções são obtidas através de grafos e hipergrafos aleatórios [36] e são ditas perfeitas por nunca levarem a colisões durante os acessos à tabela *hash*. Além disso, tais funções são mínimas, por não necessitarem de espaço adicional na tabela, e preservadoras da ordem, por manterem uma ordem qualquer especificada para as chaves. Funções com essas características são referidas na literatura pela sigla OPMPHF [1]. A principal referência usada na implementação é [47], enquanto que uma maior formalização da técnica pode ser encontrada em [19]. A Seção 2.2.2 traz maiores esclarecimentos sobre o funcionamento dessas funções e sua aplicação a este trabalho. Diferentes abordagens existem para a construção de OPMPHFs, como [23, 24].

Os modelos clássicos de recuperação de informação, entre eles o modelo lógico, o modelo vetorial e o modelo probabilístico, são tratados em referências tradicionais e modernas da área, como [83, 67, 26, 85, 6]. Técnicas para aceleração do processamento de consultas são discutidas em [32, 53]. Já Persin [57, 58] propõe alterações sobre a estrutura de arquivos invertidos, ordenando as listas de ocorrências pelas freqüências dos termos nos documentos, de forma a reduzir gastos com tempo e memória na avaliação. Tais modificações têm impacto sobre a estrutura e técnica de compressão das listas. Em [84] é apresentado um novo método para codificação de arquivos invertidos que visa reduzir as sobrecargas de descompressão para o processamento de consultas que utilizam tanto o modelo lógico como o vetorial. Na Seção 2.3 apresenta-se em maior detalhe o modelo vetorial e as estratégias para utilizá-lo com índices ordenados por freqüência no processamento eficiente de consultas

Em relação a algoritmos paralelos e distribuídos para recuperação de informação, muito do trabalho inicial com máquinas massivamente paralelas é devido a Stanfill [74, 72, 73]. O balanceamento de carga e a questão da escalabilidade de multiprocessadores simétricos para aplicações em recuperação de informação é discutido em [45]. Aspectos de desempenho de sistemas de informação distribuídos são avaliados em [13]. O processamento distribuído de consultas é abordado por Tomasic e García-Molina [80, 81] e Ribeiro-Neto e Barbosa [63, 7]. Esse último trabalho apresenta uma análise comparativa entre o desempenho dos índices local e global e nele foi baseada a arquitetura utilizada na presente dissertação, descrita na Seção 3.1.

Em [61] apresenta-se um algoritmo distribuído para a indexação de textos. Entretanto, essa abordagem é limitada por não utilizar memória auxiliar em disco. Assim, as listas invertidas devem ser acomodadas completamente em memória principal durante o processo de indexação,

[1] De *Order Preserving Minimal Perfect Hash Function*.

o que inviabiliza seu uso para grandes volumes de texto. A família de algoritmos implementada no presente trabalho é proposta em [62] e detalhadamente descrita no Capítulo 3.

Outras arquiteturas distribuídas para recuperação de informação são tratadas em [44, 46], utilizando-se réplicas parciais de coleções para aumentar a velocidade e a tolerância a falhas em sistemas de informação. Já [88, 89] discutem seleção e busca em coleções distribuídas.

Por fim, [85] e [6] são referências atuais na área de recuperação de informação e cobrem a maioria dos conceitos utilizados neste trabalho, como compressão de arquivos invertidos, *hashing* perfeito, modelos de recuperação de informação e sistemas de informação distribuídos.

1.3 Objetivos e Contribuições

Os objetivos do presente trabalho são o desenho, análise, implementação e experimentação de uma família de algoritmos para a construção distribuída de arquivos invertidos sobre grandes coleções textuais. Os algoritmos são implementados em um arcabouço único, que permite a seleção de qual estratégia de indexação deseja-se utilizar (inclusive a seqüencial), o uso ou não de compressão dos índices, entre outras funcionalidades. Como saídas, são gerados o arquivo invertido para a coleção processada e outras estruturas para acesso ao índice e auxílio ao processamento de consultas através do modelo vetorial.

A principal contribuição do trabalho é a elaboração de um mecanismo que permite a indexação eficiente de grandes volumes de texto, utilizando *hardware* de baixo custo e amplamente disponível, como estações de trabalho conectadas em rede. Além de oferecer um custo/benefício maior que a utilização de máquinas paralelas, a abordagem proposta proporciona uma melhor escalabilidade, na medida em que mais estações podem ser adicionadas à rede e cooperar na tarefa de indexação, sem degradar o desempenho dos algoritmos.

Outras contribuições incluem o processo seqüencial de indexação, que gera de forma eficiente arquivos invertidos comprimidos ordenados por freqüência, juntamente com as estruturas auxiliares para o processamento de consultas. Deve-se mencionar também a elaboração do método para ordenação linear das listas de documentos pelas freqüências dos termos, adaptado do *list radix sort* [42], bem como uma modificação sobre o processo de intercalação de arquivos invertidos que permite uma considerável redução no número de acessos a disco. Ainda no âmbito deste trabalho, foi elaborado um mecanismo de extração de texto a partir de diversos formatos, como HTML, PDF, Postscript e Microsoft Word, utilizado na máquina de busca do Projeto SIAM [70].

1.4 Notação e Organização

Nesta seção procuram-se definir os principais símbolos utilizados neste texto e esclarecer convenções de notação e conceitos. Descreve-se também a organização do restante da dissertação.

Símbolo	Significado
C	A coleção de documentos textuais
c	Número de documentos na coleção
n	Tamanho da coleção em bytes
w	Número de palavras na coleção
x	Número de pontos de indexação na coleção
$\|w_C\|$	Tamanho médio das palavras da coleção
V	O vocabulário da coleção
v	Número de termos no vocabulário
$\|w_V\|$	Tamanho médio das palavras do vocabulário
Σ	O alfabeto da coleção
σ	Número de caracteres no alfabeto
K, β	Parâmetros da Lei de Heaps [37]
θ	Parâmetro da Lei de Zipf [90]
L	Uma lista invertida
F	O arquivo invertido final
f	Tamanho do arquivo invertido final em bytes
F', F''	Índices invertidos temporários
f', f''	Tamanhos dos índices temporários em bytes

Tabela 1.1: Variáveis relativas à coleção local a cada processador.

A escolha de símbolos apropriados para representar variáveis em qualquer trabalho que envolva mais de uma área de pesquisa não é uma tarefa trivial. O número de variáveis consideradas é grande e torna-se difícil a atribuição de símbolos relevantes. Além disso, muitas vezes as referências consultadas divergem nos símbolos utilizados e estes em geral são derivados de nomes em inglês. Neste trabalho procurou-se utilizar os símbolos que melhor representassem as variáveis correspondentes, levando-se em conta seu uso mais comum na literatura e a facilidade de associação ao seu significado, expresso em inglês.

A Tabela 1.1 apresenta as variáveis relativas à coleção local a cada processador, enquanto que a Tabela 1.2 traz os parâmetros associados ao ambiente de execução dos algoritmos. Nota-se que algumas variáveis distintas possuem símbolos semelhantes, mas acredita-se que isso não levará a ambigüidade e, onde isso puder ocorrer, uma distinção explícita será feita. No Capítulo 5 as coleções e o ambiente de operação considerados neste trabalho são caracterizados e valores concretos são associados às variáveis.

Ao longo de todo o texto, refere-se a um termo qualquer da coleção por t. O i-ésimo termo é denominado especificamente t_i. Da mesma forma, um documento qualquer é referido por d, enquanto que, o j-ésimo documento, por d_j. A freqüência de um termo qualquer t em um documento d é representada por $f_{t,d}$, já a freqüência do i-ésimo termo no j-ésimo documento, por $f_{i,j}$. Por outro lado, a freqüência de um termo t em documentos da coleção, isto é, o número

Símbolo	Significado
p	Número de processadores em operação
M	Memória disponível para indexação por processador, em bytes
B	*Buffer* para armazenamento das triplas
b	Tamanho do bloco de dados (tipicamente igual a um bloco de disco)
R	Número de *runs* produzidos na indexação
λ	Constante de proporcionalidade do algoritmo linear de ordenação
t_r	Tempo médio de transferência do disco por byte
t_s	Tempo médio de *seek* do disco
t_z	Tempo médio de compressão por byte comprimido
t'_z	Tempo médio de descompressão por byte comprimido
t_p	Tempo do primeiro *parsing* do texto por byte
t'_p	Tempo do segundo *parsing* do texto por byte
t_n	Tempo de transferência pela rede por byte
t_{cl}	Tempo para comparar duas triplas $\langle t, f, d \rangle$
t_{cs}	Tempo para comparar duas *strings* por byte
t_A	Tempo total gasto pelo algoritmo de indexação A
t_{A_k}	Tempo gasto pelo algoritmo A na fase k

Tabela 1.2: Parâmetros associados ao ambiente de operação dos algoritmos.

de documentos nos quais o termo ocorre (que corresponde também ao tamanho da lista invertida de t) é representada por f_t. O tamanho em bytes de uma tripla formada por identificadores de termo e documento e a relativa freqüência é expresso por $|\langle t, f, d \rangle|$ e geralmente corresponde a 10 bytes (4 bytes para os identificadores t e d e 2 bytes para a freqüência f). Em algumas situações, entretanto, restrições de alinhamento de memória relativas a arquitetura de máquina fazem com que esse valor seja de 12 bytes.

Deve-se salientar a diferença entre $|w_C|$ e $|w_V|$, respectivamente, o tamanho médio das palavras na coleção e no vocabulário. O primeiro valor, $|w_C|$, leva em conta todas as palavras que surgem no texto e, como palavras menores são mais freqüentes, é menor que $|w_V|$, que considera apenas uma vez cada palavra distinta.

O alfabeto Σ é conjunto dos caracteres formadores de palavras. Neste trabalho, consideram-se todos os caracteres alfanuméricos (letras e dígitos), com, entretanto, algumas ressalvas descritas na Seção 2.1.

Como destacado acima, os parâmetros da Tabela 1.1 se referem à coleção local a cada processador. Quando, no ambiente distribuído, for necessária a distinção entre uma determinada variável local e uma global, utiliza-se um índice apropriado. Por exemplo, V_g refere-se ao vocabulário global, enquanto que V_3, ao vocabulário da coleção local ao processador 3.

Nota-se que o parâmetro R na Tabela 1.2 é, na verdade, dependente tanto da coleção como do ambiente, pois é resultado da relação entre o número de pontos de indexação na coleção, x,

e a memória disponível para inversão, M. As relações entre as diversas variáveis presentes nas tabelas anteriores são esclarecidas na Seção 2.1. Já o valor λ corresponde à constante de proporcionalidade do algoritmo linear para a ordenação de listas invertidas mencionado anteriormente como uma das contribuições deste trabalho.

Outro aspecto a ser notado na Tabela 1.2 é o uso de valores distintos para as taxas de compressão e descompressão de dados. As referências que incluem análises levando em conta compressão de dados em indexação [51, 85] de textos usam o mesmo valor para ambas as taxas. Entretanto, o tempo de compressão pode ser bastante reduzido através da pré-computação de alguns códigos, o que não pode ser feito para a descompressão, obrigando o uso de variáveis distintas, t_z e t'_z, para as taxas de compressão e descompressão, respectivamente.

O restante desta dissertação está organizado da seguinte forma. O Capítulo 2 descreve a construção seqüencial de arquivos invertidos, apresentando o algoritmo no qual as abordagens distribuídas serão baseadas. Nesse capítulo, além de conceitos básicos de arquivos invertidos e processamento de consultas, são discutidos aspectos como *parsing* de textos, *hashing* perfeito, compressão de índices, ordenação eficiente de listas de ocorrências e intercalação de arquivos invertidos por vários caminhos.

O Capítulo 3 apresenta a família de algoritmos distribuídos para a indexação de textos. Nele é discutida a arquitetura considerada neste trabalho, a organização global do vocabulário e do índice e as técnicas de processamento de consultas nesse ambiente. São então descritos em detalhes os três algoritmos distribuídos propostos, além da fase inicial comum a todos eles, a computação do vocabulário global.

Detalhes da implementação realizada neste trabalho são descritos no Capítulo 4. O sistema desenvolvido (denominado **DIG**, de *Distributed Index Generation*) é decomposto em seus diagramas de classes e são discutidas considerações de eficiência de implementação, organização das *threads* que realizam o trabalho de indexação, entre outros aspectos.

Os resultados analíticos e experimentais são mostrados no Capítulo 5. As coleções utilizadas nos testes são caracterizadas, assim como o ambiente de operação do sistema implementado. Os modelos elaborados nos Capítulos 2 e 3 são validados em relação aos resultados reais. Nesse capítulo são mostrados também a sintaxe e alguns exemplos de utilização do sistema.

Finalmente, no Capítulo 6 são apresentadas as conclusões deste trabalho e as contribuições proporcionadas pela abordagem adotada. Propõem-se também trabalhos futuros e melhorias ao sistema implementado.

Capítulo 2

Construção Seqüencial

No presente capítulo apresentam-se os principais conceitos relacionados à geração seqüencial de arquivos invertidos e caracteriza-se a estratégia de indexação adotada nesta dissertação. A Seção 2.1 descreve em detalhes a principal estrutura de dados deste trabalho, os arquivos invertidos. A Seção 2.2 apresenta as diversas técnicas para acesso ao vocabulário do texto, um aspecto central na construção eficiente de índices. Por fim, os modelos para recuperação de informação mais utilizados são descritos na Seção 2.3, assim como as metodologias para processamento eficiente de consultas utilizando os índices no formato apresentado na Seção 2.1.

2.1 Arquivos Invertidos

Como citado na Seção 1.1, onde se discutia a motivação deste trabalho, arquivos invertidos têm se consolidado como a técnica de indexação de textos mais popular, por sua simplicidade e eficiência. Esta seção descreve diversos aspectos relativos a tal estrutura de dados, incluindo sua definição e formas de compressão. São também feitas algumas considerações sobre transformações aplicadas ao texto para a geração de arquivos invertidos e, finalmente, as principais técnicas para a construção desses índices.

2.1.1 Definição

Índices são mecanismos relacionados à operação de **busca** (também pesquisa ou consulta) de uma dada **chave** em uma massa de dados. Neste trabalho, considera-se que a massa de dados é constituída de texto e que a chave de busca corresponde a uma ou mais palavras, representadas por cadeias de caracteres (também chamadas *strings*). Supõe-se também que o texto encontra-se dividido em porções lógicas, chamadas **documentos** (por exemplo, capítulos de um livro, registros de um processo criminal ou páginas *Web*). Entretanto, em outras aplicações, os dados poderiam consistir de arquivos de áudio e a chave seria um determinado intervalo de som, ou ainda poderia-se tratar da busca de uma impressão digital em um banco de dados policial, ou

a pesquisa de uma cadeia de DNA em um arquivo de informações genéticas. Em todas essas aplicações, índices podem auxiliar a operação de busca.

O emprego de índices não é, no entanto, universal. Se a massa de dados é pequena, a busca direta (ou *on-line*) da chave pode ser rápida o suficiente para descartar a necessidade de uso de estruturas auxiliares para acelerar o processo. Além disso, se os dados a serem pesquisados são muito voláteis, isto é, mudam com freqüência, ou se não há espaço adicional disponível, a busca *on-line* é a única alternativa. Em um grande número de aplicações, porém, os dados são estáticos, tais como livros e enciclopédias, ou semi-estáticos, como processos médicos e jurídicos ou coleções de páginas *Web*, atualizados somente em intervalos fixos de tempo.

No caso de textos, há vários índices que permitem a realização eficiente da operação de localização de uma chave, como árvores PATRICIA [54], arranjos de sufixos [48], arranjos PAT [30], arquivos invertidos [35], arquivos de assinatura [22] e mapas de bits (*bitmaps*) [15]. Da mesma forma, há várias maneiras de se definir que tipo de localização se deseja obter a partir do índice (a chamada **granularidade** do índice): posições exatas (bytes dentro do texto), documentos ou blocos de texto. Algumas estruturas são mais apropriadas a um tipo ou outro de granularidade.

Dentre as diversas estruturas de indexação, os arquivos invertidos são os mais utilizados, pela simplicidade de sua estrutura, eficiência nas buscas, adequação a vários tipos de granularidade do índice e possibilidade de compressão. Esta última propriedade faz com que esses índices ocupem um pequeno espaço extra em relação ao tamanho do texto. Arquivos invertidos são classificados como um índice orientado a palavras, pois essas são as menores unidades de informação consideradas no texto. Não é possível, por exemplo, recuperar diretamente as ocorrências de parte de uma palavra (ou de uma frase contendo várias palavras) em um arquivo invertido. Outras estruturas, como os arranjos de sufixos, indexam todos os caracteres do texto, permitindo qualquer tipo de consulta, ao custo de um maior espaço extra.

Como mencionado na Seção 1.1, um arquivo invertido é composto por subestruturas:

- o **vocabulário** ou **léxico**, correspondente à lista de todos os termos distintos que ocorrem no texto. Esta estrutura geralmente é pequena em relação ao texto e pode ser armazenada completamente em memória principal. Há ainda técnicas para redução do espaço ocupado pelo vocabulário em memória, discutidas na Seção 2.2.

- as **listas de ocorrências** ou **listas invertidas**, que indicam, para cada termo do vocabulário, as posições do texto onde tal termo ocorre. A definição de "posição" determina a granularidade do índice.

Tradicionalmente, a denominação "arquivo invertido" aplica-se aos índices que apontam para documentos. Estruturas com outras granularidades, como bytes individuais [2] ou blocos de tamanho fixo [49, 5], são chamadas "índices invertidos". Entretanto, pode-se considerar

Documento	Texto
1	a onda anda
2	aonde anda
3	a onda?
4	a onda ainda
5	ainda onda
6	ainda anda
7	aonde?
8	aonde?
9	a onda a onda

Tabela 2.1: Exemplo de uma coleção textual simples.

a indexação a nível de documentos como mais uma alternativa para a granularidade e, neste trabalho, refere-se a arquivos ou índices invertidos indistintamente.

Na Tabela 2.1 é apresentado um pequeno texto, no qual cada verso (linha) da poesia[1] foi considerado um documento, formando assim uma pequena coleção que será utilizada para fins de exemplificação ao longo deste trabalho. Nota-se que todas as letras foram convertidas para minúsculas, de forma a evitar a diferenciação indevida entre palavras. Como será justificado na Seção 2.1.2, uma operação geralmente realizada sobre as palavras de um texto ao se construir um arquivo invertido é a conversão de todas as suas letras para minúsculas ou maiúsculas.

Uma representação esquemática de um arquivo invertido tradicional construído sobre o texto da Tabela 2.1 é exibida na Figura 2.1. Nela, a estrutura correspondente ao vocabulário é representada ao lado esquerdo, contendo os termos distintos do texto. Cada entrada do vocabulário aponta para a lista invertida do termo correspondente. Cada posição dessa lista contém, por sua vez, um par $\langle d, f \rangle$, indicando o número de um documento d onde o termo ocorre e a freqüência f do termo nesse documento. As listas encontram-se ordenadas da forma mais natural, pelos números dos documentos nos quais os termos ocorrem.

A Figura 2.2 apresenta um arquivo invertido para a mesma coleção, porém com as listas de ocorrências ordenadas pelas freqüências dos termos nos documentos, o formato considerado neste trabalho. Os componentes $f_{t,d}$, isto é, as freqüências de cada elemento das listas foram representados em primeiro lugar nas células, para destacar que eles são a chave primária para ordenação das listas. Pares com o mesmo elemento $f_{t,d}$ são ordenados pela chave secundária, o número do documento d. Nota-se que a única diferença dessa estrutura para aquela apresentada na Figura 2.1 é a ordenação das listas; ambos os índices possuem os mesmos elementos, pois têm a mesma granularidade, e ocupam o mesmo espaço (sem se considerar o emprego de compressão).

[1] *A Onda*. Manuel Bandeira. Estrela da vida inteira. 4ª ed. J. Olympio, 1973, p. 286.

Já a Figura 2.3 apresenta um índice com uma granularidade diferente, para o mesmo texto. Ele aponta para o endereço em bytes, ou *offset*, do início de cada ocorrência de um termo no texto. Esse tipo de índice não necessita assumir nenhuma estrutura sobre o texto sendo indexado, como parágrafos, capítulos ou documentos, e pode ser utilizado para acelerar a busca (exata ou aproximada) de padrões (de uma ou mais palavras). As buscas são realizadas eficientemente, bastando-se acessar o vocabulário e o índice, sem recorrer ao texto. Esse paradigma é utilizado em [2] para implementar o sistema IGREP de busca aproximada. Aspectos analíticos dessa abordagem são discutidos em [3]. Cumpre notar que esse índice possui mais ponteiros, portanto, demanda mais espaço que os outros apresentados anteriormente.

Outra possibilidade para a granularidade de índice é exibida na Figura 2.4, que traz um arquivo invertido relativo a blocos do texto. Os índices por *offsets* e por documentos podem representar uma sobrecarga de espaço considerável em relação ao texto original. Assim, indexar blocos de tamanho fixo ou um número pré-determinado de blocos pode ser uma alternativa para reduzir o tamanho do índice. Torna-se necessária, entretanto, uma fase posterior de busca no texto, para se determinar a ocorrência exata do termo pesquisado. No exemplo da Figura 2.4, o texto foi dividido em quatro blocos, correspondentes aos documentos [1,2], [3,4], [5,6] e [7,8,9]. Os apontadores para blocos foram representados em binário, para destacar que apenas 2 bits são necessários; nota-se, portanto, a economia de espaço proporcionada por esse índice.

O sistema GLIMPSE [49] utiliza esta última abordagem, criando um índice que divide o texto em 256 blocos de igual tamanho, o que permite que cada apontador possua apenas um byte. O índice é tão compacto que o sistema é bastante popular para buscas em diretórios pessoais, ficando o índice armazenado na própria área em disco do usuário. Buscas exatas e aproximadas são suportadas, contudo torna-se necessária uma pesquisa direta no texto para localizar a ocorrência precisa do padrão, uma arquitetura que convencionou-se chamar **busca em dois níveis** [87]. O desempenho é aceitável para textos de até 200 Mbytes, a partir dos quais o tempo de resposta passa a ser muito alto em relação a outros índices. Técnica semelhante é utilizada em [5] e em [55], considerando-se, porém, blocos com número fixo de palavras.

Como último ponto desta seção, apresentam-se duas leis empíricas que relacionam o comportamento das duas estruturas componentes de arquivos invertidos (vocabulário e listas de ocorrências) às propriedades do texto. A primeira, a Lei de Heaps [37], é bastante precisa e determina o número de palavras no vocabulário de um texto. Já a Lei de Zipf [90] é mais ampla, imprecisa e caracteriza a distribuição das palavras, seu comprimento, freqüência em documentos, entre outras propriedades.

A Lei de Heaps estabelece que o vocabulário de um texto de n bytes tem tamanho $v = Kn^{\beta}$, onde K e β são parâmetros dependentes do texto. O valor de K é normalmente entre 5 e 50, enquanto que β é um valor positivo menor que 1. Experimentos [2, 3, 5] mostram que, para a coleção TREC, o valor de K é 4.8 (apresentando variações consideráveis quando se toma uma sub-coleção específica da TREC) e β fica sempre entre 0.4 e 0.6. Isso comprova que o

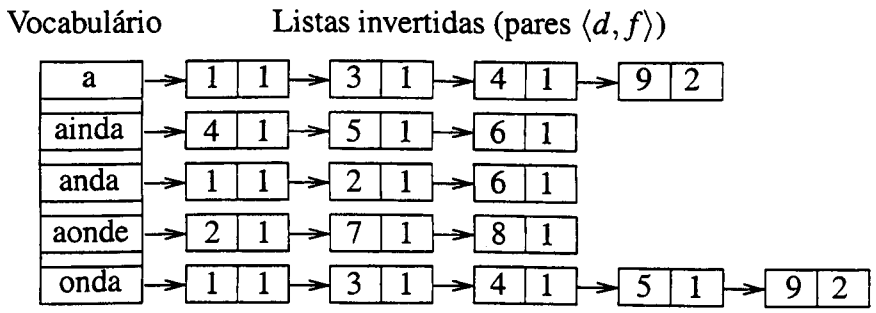


Figura 2.1: Arquivo invertido tradicional para o texto da Tabela 2.1.

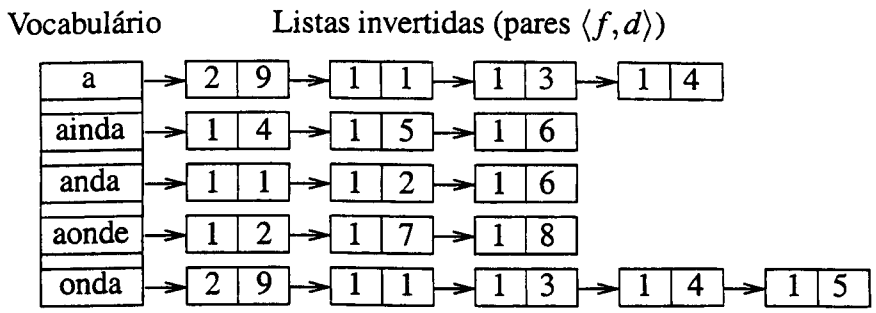


Figura 2.2: Arquivo invertido ordenado por freqüência para o texto exemplo.

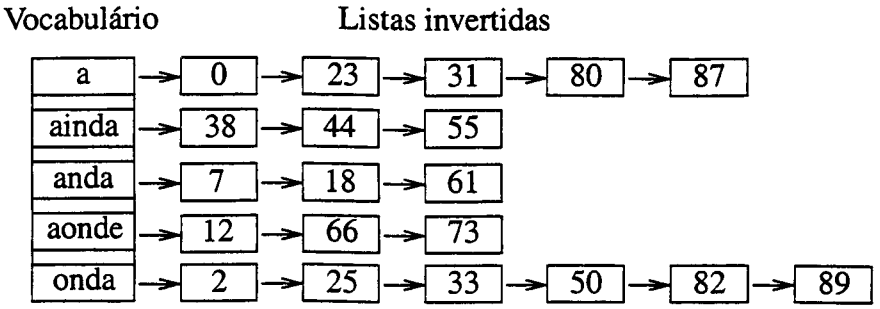


Figura 2.3: Arquivo invertido por *offsets* para o texto exemplo.

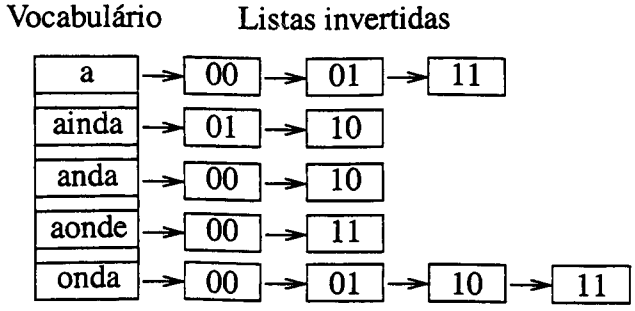


Figura 2.4: Arquivo invertido por blocos para o texto da Tabela 2.1.

vocabulário cresce sublinearmente com o texto, em uma proporção próxima a sua raiz quadrada, ou seja, $v = O(\sqrt{n})$. Apesar do número de palavras distintas em uma dada linguagem ser fixa, pode-se considerar que o vocabulário continua crescendo indefinidamente com o texto, devido a erros de digitação, uso de palavras estrangeiras, termos técnicos ou novos vocábulos.

A Lei de Zipf é um modelo aproximado para a distribuição das ocorrências dos termos distintos em um texto. Ela diz que a freqüência da i-ésima palavra mais comum é $1/i^{\theta}$ vezes a freqüência da palavra mais comum. Isso implica que, em um texto de n bytes com v palavras, o número de vezes que a i-ésima palavra mais freqüente ocorre é dado por

$$\frac{n}{i^{\theta} H_v(\theta)},$$

onde $H_v(\theta)$ é o número harmônico de ordem θ, definido por

$$H_v(\theta) = \sum_{j=1}^{v} \frac{1}{j^{\theta}}.$$

O valor de θ é dependente do texto e vale entre 1.7 e 2.0 para amostras da coleção TREC. Com isso, $H_v(\theta) = O(1)$, mostrando que a distribuição das ocorrências das palavras no texto é bastante tendenciosa, com poucas delas ocorrendo muitas vezes e muitas aparecendo um número pequeno de vezes. Um reflexo desse comportamento é a distribuição do tamanho das listas invertidas (os valores f_t), como será mostrado na Figura 5.6: um número pequeno de palavras freqüentes (geralmente, *stopwords*) possui listas com muitos elementos, enquanto que a maioria dos termos possui listas com poucos elementos (tipicamente, apenas 1). A Lei de Zipf influencia também a distribuição das freqüências de termos dentro dos documentos (os valores $f_{t,d}$), de acordo com a Figura 5.13: dada uma lista invertida, na maioria dos documentos dessa lista, o termo ocorre poucas vezes (em grande parte, apenas uma vez), ao passo que em poucos documentos nos quais ele aparece um número maior de vezes.

2.1.2 Operações sobre o texto

A eficácia de arquivos invertidos na recuperação da informação solicitada pelo usuário é totalmente dependente da definição e da seleção dos termos (palavras) a serem indexados, uma vez que estes comporão o vocabulário e determinarão quais listas serão processadas durante as consultas. A abordagem mais intuitiva para a definição das palavras de um texto é considerar toda cadeia composta por caracteres alfanuméricos, isto é, letras e dígitos. Os dígitos devem estar presentes de forma a se permitir consultas contendo datas, números de páginas ou seções, versões de *software*, etc.

Todavia, essa estratégia simplista sofre de algumas limitações. Em primeiro lugar, ela irá distingüir indevidamente palavras que diferem apenas em relação a letras maíusculas ou

minúsculas. Assim, caso uma palavra apareça no início e no meio de uma frase, duas listas distintas serão criadas para ela, uma para a primeira ocorrência (que possui a primeira letra em maiúsculo) e outra para a ocorrência com todas as letras em minúsculo. Da mesma forma, caso a consulta seja especificada com letras em maiúsculo, ela pode não encontrar nenhum casamento no índice.

Outro aspecto desfavorável dessa abordagem é o fato de que algumas buscas devem ignorar os sufixos das palavras, considerando apenas o radical. Por exemplo, um usuário que busca por "*trabalho*" e "*computação*" pode estar interessado em documentos que mencionem "*trabalhador*" e "*computador*", ou ainda "*trabalhos*" e "*computacionais*". O processo de se removerem os sufixos de uma palavra, reduzindo-a a seu radical, é chamado *stemming*. Através dele, os termos do texto e da consulta mencionada seriam convertidos a "*trabalh*" e "*comput*", possibilitando que todas as ocorrências relacionadas fossem recuperadas.

Ainda assim, nem todos os termos de um documento representam igualmente sua semântica. Em geral, substantivos carregam maior significado, ao contrário de artigos, preposições e conjunções. Estas últimas classes de palavras aparecem quase na totalidade dos documentos de uma coleção e não consitituem bons discriminadores de relevância. Assim, uma prática comum é simplesmente ignorar essas palavras, chamadas na literatura de *stopwords*. Um benefício importante da não-indexação de *stopwords* é a redução do índice. Como elas ocorrem em parte significativa dos documentos, suas listas de ocorrências são grandes e, com sua remoção, o tamanho do índice pode ser reduzido em até 40% [2].

Descrevem-se agora os aspectos negativos dessas três operações e como elas se aplicam ao presente trabalho. Primeiramente, a não-diferenciação entre caracteres maiúsculos e minúsculos introduz falsos casamentos quando se procuram nomes próprios (ou siglas) que têm a mesma grafia de nomes comuns. Entretanto, as vantagens proporcionadas por essa conversão são superiores às desvantagens e, neste trabalho, adota-se a conversão de todas as letras para minúsculas durante a indexação. Também consideram-se os dígitos na formação de palavras, adicionando-se, no entanto, a restrição de se permitir no máximo quatro dígitos por termo, como sugerido em [85, Capítulo 3]. Dessa forma, evita-se o crescimento demasiado do vocabulário (com a inclusão de muitos números sem relevância) e permite-se a realização de buscas por datas, a maioria das versões de *software* e números em geral.

A operação de *stemming*, por seu lado introduz falsos casamentos, pois converte um conjunto de palavras para um mesmo radical comum. Em casos onde o usuário deseja buscar exatamente a palavra pesquisada, como na busca por frases, resultados indesejados serão retornados. As desvantagens são maiores em coleções contendo grandes volumes de texto (pois a probabilidade de falsos casamentos é maior), com muitos erros de digitação (pois os radicais são extraídos de forma incorreta), heterogêneas (a variedade de temas combinada a *stemming* pode levar a maiores ambigüidades) ou multilingüais (os algoritmos para remoção de sufixos variam muito entre diferentes línguas). A *Web* possui todas essas características, razão pela

qual as máquinas de busca geralmente não utilizam *stemming*. Por tais motivos, não se emprega *stemming* neste trabalho e as palavras são consideradas da forma em que foram digitadas.

A eliminação de *stopwords* é ainda mais controversa. A própria caracterização de uma palavra como *stopword* é questionável. Palavras que em determinado contexto não carregam nenhum significado podem ser representativas em outros. Artigos e preposições, por exemplo, podem constituir parte de um nome próprio (como o título de um filme ou música), de forma que o usuário pode desejar buscar uma frase contendo uma ou mais dessas classes de palavras. Caso elas não sejam indexadas, o sistema deve retornar uma resposta incompleta ou esquadrinhar todo o texto, o que pode ser caro. Além disso, o fato de palavras comuns não representarem bons indicadores de relevância é levado em conta pelo modelo vetorial, que impede palavras que ocorrem em muitos documentos de influenciarem na ordenação das respostas. O argumento de que a eliminação de *stopwords* reduz o tamanho dos índices é amenizado pela compressão: suas listas de ocorrências contém muitos valores similares e podem ser compactadas a taxas mais altas, segundo mostrado na Seção 2.1.3 a seguir, reduzindo sua proporção no índice comprimido.

No presente trabalho, portanto, consideram-se como palavras as seqüências de caracteres alfanuméricos, permitindo-se no máximo quatro dígitos por palavra e convertendo-se as letras para minúsculas. Estabelece-se um tamanho máximo de 256 caracteres por palavra, o que simplifica considerações de implementação (esclarecidas na Seção 4.3.2) e funciona como uma barreira para casos degenerados: uma cadeia com mais de 256 caracteres não deve representar nenhuma palavra significativa e, logo, não deve ser considerada. Não se utiliza *stemming* e não se eliminam *stopwords*, convenções similares às adotadas na maioria das máquinas de busca para *Web*. Essa estratégia se justifica pelo fato de se propor aqui a indexação tanto de coleções estruturadas (como a TREC [33]), como de coleções heterogêneas (como as páginas coletadas para o projeto SIAM [70]). Mais detalhes sobre quais operações podem ser aplicadas sobre textos são encontrados em [91].

2.1.3 Compressão

Apesar da simplicidade da estrutura exibida nas Figuras 2.1 a 2.4, arquivos invertidos podem consumir espaço considerável: 50 a 100% do tamanho do texto original. Considerando o banco de dados exemplo, o texto possui 94 bytes, contando-se espaços, quebras de linha, pontuação e o marcador de fim-de-arquivo, além das palavras. Se, na representação do arquivo invertido da Figura 2.1, forem usados inteiros de 4 bytes para os números dos documentos e de 2 bytes para as freqüências, cada ponteiro consumirá 6 bytes. Como têm-se 18 ponteiros, o arquivo invertido ocuparia $18 \times 6 = 108$ bytes, logo, mais que o próprio texto.

Entretanto, essa representação para arquivos invertidos, embora intuitiva, carrega grande redundância de informação e pode ser comprimida através de técnicas discutidas a seguir. A chave para a compressão de arquivos invertidos é observar que os ponteiros para as ocorrências

de um termo podem ser representados através de seqüências crescentes de inteiros. No caso mais especial do índice da Figura 2.2, para cada freqüência distinta, a lista de documentos está disposta em ordem crescente e também pode ser representada através de uma seqüência.

Assim, a lista de ocorrências do termo "onda" no índice por *offsets* da Figura 2.3, corresponde à seqüência

$$\langle 2, 25, 33, 50, 82, 89 \rangle.$$

Em geral, qualquer lista de ocorrências possui a forma $\langle d_1, d_2, \ldots, d_{f_t} \rangle$, onde f_t é o tamanho da lista invertida para o termo t e d_j, $1 \leq j \leq f_t$, com $d_j < d_{j+1}$, são as ocorrências de t, que podem consistir em apontadores para documentos, *offsets* ou blocos.

Como a lista de ocorrências encontra-se sempre em ordem crescente e todo o processamento é sempre realizado a partir do início da lista, cada elemento pode ser representado pela diferença entre ele próprio e seu antecessor. Assim, cada lista é formada por um elemento inicial e uma série de **intervalos-*d***. A lista exemplificada acima seria codificada como

$$\langle 2, 23, 8, 17, 32, 7 \rangle.$$

Para se codificarem os novos valores obtidos pela substituição por intervalos, consideram-se códigos para a representação de números inteiros positivos. Se um valor x, para o qual é conhecido um limite superior X, deve ser codificado, então um código binário simples de $\lceil \log X \rceil$ bits pode ser utilizado. Essa codificação, porém, não proporciona economias em espaço para a representação por intervalos em relação à por ponteiros. Se c é o valor do maior ponteiro da coleção considerada, o maior intervalo que potencialmente pode surgir é o mesmo c, no caso de um termo que ocorre apenas na primeira e na última posição da coleção. Assim, uma codificação binária comum demandaria $\lceil \log c \rceil$ bits por ponteiro, o que geralmente não proporciona ganhos significativos em espaço.

A principal modificação proporcionada pelo uso de intervalos é que os ponteiros passam a ser constituídos por valores menores. É interessante, portanto, considerar codificações em que números menores sejam representados por menos bits. O código mais simples satisfazendo esse critério é o **unário**, que representa um inteiro x por $x-1$ bits "1", seguidos de um bit "0". Logo, o código para um número x possui x bits.

Outros códigos de comprimento variável foram considerados por Elias [21]. O código **Elias-**γ representa um número x através de um código unário para $1 + \lfloor \log x \rfloor$ seguido de $\lfloor \log x \rfloor$ bits representando $x - 2^{\lfloor \log x \rfloor}$ em binário. O código γ para um inteiro x conterá, portanto, $1 + 2\lfloor \log x \rfloor$ bits. Outro código descrito é **Elias-**δ, que codifica o prefixo $1 + \lfloor \log x \rfloor$ em Elias-γ em vez de unário, seguido da mesma representação de $x - 2^{\lfloor \log x \rfloor}$ em binário. O código δ para um inteiro x demanda $1 + 2\lfloor \log \log 2x \rfloor + \lfloor \log x \rfloor$ bits.

Há ainda outro método, o código de **Golomb** [29], que difere dos anteriores por ser pa-

Método	Intervalos-d	$f_{d,t}$
Unário	—	1.71
Binário	21.00	—
Elias-γ	6.76	1.79
Elias-δ	6.45	2.01
Golomb	6.11	—
Melhor	5.59	1.63

Tabela 2.2: Taxas de compressão para a coleção TREC, em bits por ponteiro.

rametrizado, o que permite a codificação de diferentes listas invertidas com diferentes valores para o parâmetro. Isso leva a codificação por Golomb a propocionar melhor compressão que os códigos de Elias, mas requer o conhecimento prévio das freqüências dos termos. Inicialmente, o parâmetro b é estabelecido, sendo $b = 0.69(c/f_t)$ o valor mais apropriado para compressão de índices, onde c é o valor do maior ponteiro da coleção e f_t, o número de ponteiros na lista de t. Dado b, um intervalo-d de tamanho x é codificado por $q+1$ em unário (onde $q = \lfloor (x-1)/b \rfloor$), seguido de $r = (x-1) - q \times b$ em binário. O código demanda, portanto, $\lfloor \log b \rfloor$ ou $\lceil \log b \rceil$ bits.

Para a compressão das freqüências $f_{d,t}$, que são geralmente baixas e seguem uma distribuição de Zipf bastante tendenciosa, discutida na Seção 5.3.2, Elias-γ é um código adequado. A Tabela 2.2, retirada de [51], mostra a eficiência de compressão dos vários códigos para um arquivo invertido construído sobre a coleção TREC, com ponteiros correspondentes a documentos e listas ordenadas pelos identificadores dos documentos. A linha "Melhor" indica a melhor compressão obtida em [52], através de um código de Huffman [39]. Há ainda outros métodos mais sofisticados para a compressão de listas invertidas, descritos em [85]; entretanto, como será discutido na Seção 2.1.4 a seguir, os códigos de Elias são os mais apropriados aos propósitos deste trabalho.

2.1.4 Construção

O processo de construção de um índice para um dado texto, também conhecido como **inversão**, é relativamente simples para coleções pequenas. Uma tarefa comum em cursos de algoritmos e estruturas de dados é a construção de "referências cruzadas" para pequenos textos. Isso envolve em geral o uso de algumas estruturas básicas, como árvores binárias ou tabelas *hash*, na implementação de uma estrutura de dicionário para conter as palavras do texto, e listas encadeadas, na implementação das listas de ocorrências. Embora simples enquanto a memória principal da máquina comporta todas essas estruturas, o processo de inversão torna-se complexo quando o texto cresce e o uso de memória secundária passa a ser necessário. Nesta seção discutem-se algumas técnicas para a inversão de grandes textos e argumenta-se por que alternativas intuitivas não apresentam desempenho satisfatório. Por fim, detalha-se a abordagem adotada neste

trabalho.

Inversão baseada em memória

A forma mais simples de se construir um índice para as palavras de um texto é criar uma estrutura de dicionário em memória (uma árvore binária ou uma tabela *hash*), onde cada entrada aponta para a lista de ocorrências, também armazenada em memória, da palavra correspondente. Além de simples, essa técnica é bastante rápida e imbatível quando a coleção de texto é pequena o suficiente para que todas as listas de ocorrências caibam em memória. Uma forma simples de se utilizar essa mesma técnica para a indexação de maiores volumes de texto é considerar uma grande quantidade de memória virtual e permitir que o próprio sistema operacional gerencie quais listas ficarão em memória e quais serão movidas para o disco.

Entretanto, o acesso às listas invertidas é inerentemente aleatório e, como elas podem estar armazenadas em memória secundária, também o padrão de acesso a disco torna-se aleatório. Isso causa uma grande degradação no tempo de execução do algoritmo, pois cada acesso a uma posição aleatória do disco implica em uma operação de movimentação da cabeça de leitura (ou *seek*), que é cara. Em [85], projeta-se o tempo de inversão para esta abordagem, permitindo que as listas invertidas sejam armazenadas em memória virtual e considerando-se uma coleção hipotética de 5 Gbytes, em um modelo computacional semelhante ao utilizado neste trabalho. Esse tempo é estimado em 1.100 horas, ou cerca de um mês e meio. Conclui-se, portanto, que a abordagem mais intuitiva é impraticável para maiores volumes de dados.

Inversão ordenada em disco

A grande falha da inversão baseada em memória ao utilizar armazenamento secundário é fazê-lo de forma aleatória, levando a um número grande de *seeks* ao disco. Uma forma mais racional de fazê-lo deveria priorizar os acessos seqüenciais a disco, evitando ao máximo operações aleatórias. Os algoritmos baseados em ordenação procuram explorar essa característica e funcionam da forma descrita a seguir.

À medida que o texto é processado, armazena-se em uma área da memória principal as ocorrências de cada termo da coleção. Sempre que essa área se esgota, as ocorrências são ordenadas e gravadas em um arquivo temporário. Cada pedaço ordenado desse arquivo temporário é chamado *run*. Terminado o processamento do texto, os *runs* são lidos e intercalados, gerando o arquivo invertido final. Um processo simples de intercalação toma os *runs* dois a dois, compara seus elementos e grava em uma outra área em disco o novo *run* produzido pela intercalação dos outros dois. Assim, se R *runs* são gerados durante o processamento do texto, então $\lceil \log R \rceil$ passadas sobre o arquivo temporário serão suficientes para intercalá-los e gerar o índice final.

A abordagem adotada

A técnica de inversão baseada em ordenação pode ainda ser significativamente melhorada. Em primeiro lugar, muito espaço temporário em disco é utilizado, visto que duas cópias do índice são necessárias durante a fase de intercalação. Além disso, os arquivos temporários devem conter, para cada ponto de indexação, não somente o apontador para a ocorrência, mas também o índice do termo ao qual ele se refere, o que significa 4 bytes a mais para cada apontador. Assim, uma grande quantidade de dados é transferida entre memória e disco, posto que o tamanho dos arquivos temporários é próximo ao da coleção e que, em cada uma das $\lceil \log R \rceil$ passadas da intercalação, um arquivo é lido e outro é gravado em disco. Outro ponto que deve ser abordado com cuidado é a ordenação das ocorrências. Como elas seguem uma distribuição tendenciosa, algoritmos genéricos como o *Quicksort* podem ter um desempenho desfavorável, quando comparados com métodos que exploram algum aspecto dessa distribuição.

No restante desta seção, descrevem-se as técnicas utilizadas para eliminar os pontos fracos da inversão baseada em ordenação e como elas se aplicam à abordagem adotada neste trabalho. Basicamente, emprega-se compressão de inteiros (como descrito na Seção 2.1.3) e intercalação por vários caminhos para reduzir o volume de dados transferido pelo disco, assim como uma técnica linear para a ordenação das ocorrências, além de outras melhorias.

Visão geral do processo. Os índices construídos neste trabalho são arquivos invertidos contendo apontadores para documentos e com listas de ocorrências ordenadas pela freqüência do termo no documento correspondente, assim como o índice da Figura 2.2, na página 13. Apresenta-se agora uma visão geral do algoritmo de construção seqüencial empregado; detalhes de implementação são dados no Capítulo 4.

O processo de indexação utilizado é constituído por duas fases principais, em cada qual o texto é lido uma vez. A primeira tem por objetivo identificar o vocabulário da coleção e gerar uma nova estrutura para o acesso a seus termos, enquanto que a segunda corresponde à geração propriamente dita do índice invertido. Na primeira fase, o texto é lido do disco e sobre ele é realizado um processo de *parsing*, que identifica as palavras e os limites entre documentos. As palavras distintas são armazenadas em uma estrutura de dicionário (neste caso, uma tabela *hash* tradicional, descrita na Seção 2.2.1) que conterá, ao fim do processamento do texto, o vocabulário da coleção. O vocabulário é então ordenado lexicograficamente e sobre ele executa-se um algoritmo para a geração de uma nova estrutura de acesso, uma tabela *hash* perfeita (apresentada na Seção 2.2.2). Essa nova estrutura permite a identificação de cada termo do vocabulário por um inteiro único t, o que será útil na segunda fase da indexação.

Pode-se questionar a necessidade de uma fase em separado somente para o levantamento do vocabulário, o que implica em mais uma leitura de todo o texto e poderia ser feito durante a geração do índice. A justificativa para a existência dessa fase inicial é que os algoritmos distribuídos necessitam identificar os termos por inteiros únicos já no início da fase de indexação, o

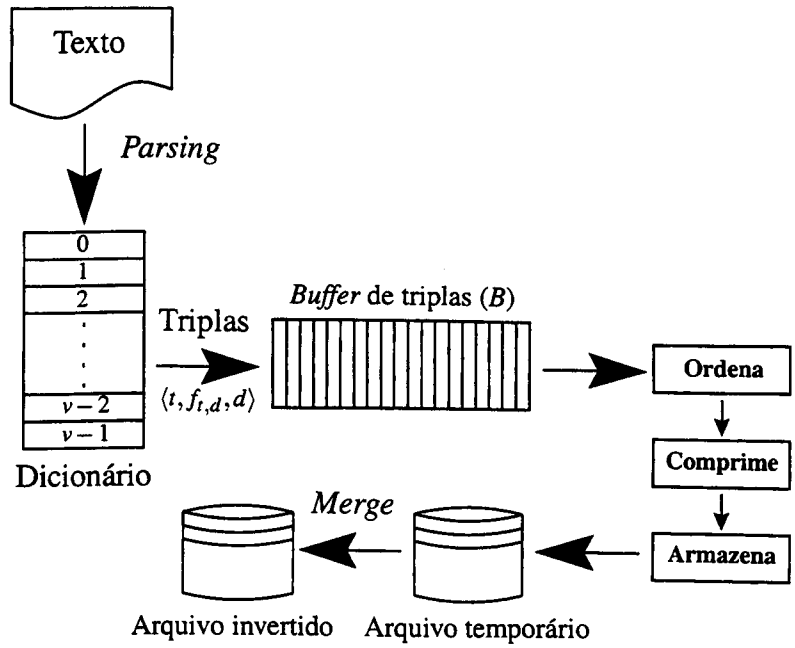


Figura 2.5: Fase de indexação do processo de geração seqüencial de arquivos invertidos.

que será esclarecido na Seção 3.2. Deve-se notar, entretanto, que ela pode levar à diminuição do tempo da indexação seqüencial como um todo.

Isso ocorre porque, em primeiro lugar, a fase de levantamento do vocabulário realiza um processamento bastante simples sobre o texto, efetuando o *parsing* das palavras e inserindo-as em uma tabela *hash* normal. Assim, ela é bastante rápida, sendo dominada quase que totalmente pelo tempo de leitura do texto, como será mostrado na Seção 5.3.1. Além disso, com o uso da tabela *hash* perfeita na segunda fase, o acesso ao vocabulário torna-se mais rápido, pois não há colisões na tabela. Essa tabela também dispensa o armazenamento das *strings* dos termos, o que reduz o espaço ocupado pelo vocabulário e permite que se utilize mais memória para as listas invertidas durante a indexação. Esses fatores amenizam o custo representado pela fase inicial e podem contribuir para que o processo de indexação como um todo seja mais rápido.

A segunda fase do processo é a geração do índice propriamente dito. Ela encontra-se ilustrada na Figura 2.5 e é descrita a seguir. O texto é novamente lido do disco, realizando-se um novo *parsing*, identificando palavras e documentos. Para cada palavra encontrada, acessa-se o dicionário construído na fase anterior, obtendo-se um identificador inteiro t. Reporta-se então a ocorrência do termo t no documento atual d, juntamente com a freqüência $f_{t,d}$. As ocorrências contidas no dicionário são, a um dado momento, copiadas para o *buffer* B, que armazena as triplas $\langle t, f_{t,d}, d \rangle$ e ocupa a maior parte da memória disponível para a indexação.

Quando o *buffer* de triplas torna-se cheio, ele é ordenado pelo componente t, em ordem crescente. Triplas com o mesmo componente t são ordenadas por $f_{t,d}$, em ordem decrescente.

Quando também os $f_{t,d}$ são iguais, ordenam-se as triplas pelo componente d, em ordem crescente. Esse processo de ordenação é melhor descrito na Seção 4.4.2, na página 86. As triplas são então comprimidas usando os códigos de Elias apresentados na Seção 2.1.3 e gravadas em um arquivo temporário em disco. Retoma-se então o processamento do texto, preenchendo-se novamente o *buffer* de triplas, agora vazio. Cada conjunto de dados ordenados, comprimidos e gravados em disco é chamado, assim como na inversão por ordenação em disco, um *run*.

A compressão das triplas que formam um *run* é realizada da seguinte forma. No início de um *run* (ou de um bloco de dados, como descrito na Seção 4.4.3), uma tripla $\langle t_{i_1}, f_{i_1,j_1}, d_{j_1} \rangle$ é codificada através de $\langle t_{i_1}+1, f_{i_1,j_1}, d_{i_1}+1 \rangle$, usando-se Elias-$\gamma$ para os componentes t e f e Elias-δ para o componente d. Os valores t_{i_1} e d_{i_1} são codificados como $t_{i_1}+1$ e $d_{i_1}+1$ devido à contagem de termos e documentos iniciar do zero e os códigos de Elias representarem apenas números positivos. Uma tripla $\langle t_{i_2}, f_{i_2,j_2}, d_{j_2} \rangle$ seguinte é assim codificada: t_{i_2} é representado por $t_{i_2} - t_{i_1} + 1$; se $t_{i_2} = t_{i_1}$, então f_{i_2,j_2} é substituído por $f_{i_1,j_1} - f_{i_2,j_2} + 1$ (já que $f_{i_1,j_1} \geq f_{i_2,j_2}$). Finalmente, se também $f_{i_1,j_1} = f_{i_2,j_2}$, d_{j_2} é representado por $d_{j_2} - d_{j_1}$, pois d_{j_2} será sempre maior (nunca igual) que d_{j_1}. A codificação por intervalos é, portanto, utilizada sempre que possível em todos os componentes das triplas.

Na compressão dos componentes d, não se pode utilizar o código de Golomb, citado na seção anterior, pois esse tipo de código não é adequado para a compressão de listas invertidas ordenadas por freqüência. O código de Golomb é parametrizado e baseia-se no conhecimento do número de documentos em uma determinada lista (o valor f_t) para ajustar o parâmetro b de forma a comprimir da melhor forma possível aquela lista (por isso ele também é classificado como um método local). Entretanto, para listas ordenadas por freqüência, as seqüências de documentos são reiniciadas a cada nova freqüência, e não mais a cada novo termo. Com isso, é impossível prever quantos documentos há por trecho da lista e, logo, não se pode estabelecer o parâmetro b, a chave para a compressão por Golomb.

No momento em que todo o texto foi processado, passa-se à intercalação, ou *merge*, dos *runs* produzidos. Apesar de ser parte da segunda fase de construção do índice, pode-se considerar a intercalação como uma etapa distinta do processo, por ela se diferenciar da anterior, na qual o texto é processado e os *runs* são produzidos. Se R *runs* foram gerados pela fase de processamento do texto, uma estrutura de fila de prioridades (por exemplo, um *heap* [69, 16]) com R posições pode ser utilizada para intercalar os *runs* e produzir o arquivo invertido final. Este último não contém triplas, mas sim duplas $\langle f_{t,d}, d \rangle$, correspondendo aos elementos das listas invertidas de cada termo distinto da coleção. O endereço de início e o tamanho de cada lista são armazenados em separado, permitindo acesso às ocorrências de cada termo individualmente.

Em [51, 85] é discutida uma modificação sobre a técnica de intercalação que permite a geração do índice final sobre a área do próprio arquivo temporário, evitando gastos com espaço extra. Essa técnica, chamada intercalação *in-place*, acrescenta outros custos à indexação, pois deve reordenar blocos no arquivo final, e não é adequada à geração distribuída. Além disso, o

custo do armazenamento secundário hoje é baixo o bastante para que as estações de trabalho disponham de espaço livre em disco para uma cópia temporária do arquivo invertido.

Representação algorítmica. O processo de indexação seqüencial descrito acima é melhor formalizado pelo Algoritmo 2.1. Nele, a intercalação dos *runs* é considerado como uma fase distinta.

Fase A: A1) **para cada** $d_j \in C$ **faça**
A1.1) *Leia*(d_j)
A1.2) **para cada** $t_i \in d_j$ **faça**
$V \leftarrow V \cup \{t_i\}$
fim para
fim para
A2) *Ordene*(V)
A3) $h_{perf} \leftarrow$ *ConstruaHashingPerfeito*(V)
$B \leftarrow \emptyset$ /* Aloca e inicializa *buffer* */
$R \leftarrow 0$ /* Inicializa número de *runs* */

Fase B: B1) **para cada** $d_j \in C$ **faça**
B1.1) *Leia*(d_j)
B1.2) **para cada** $t_i \in d_j$ **faça**
$B \leftarrow B \cup \langle t_i, d_j, f_{i,j} \rangle$
fim para
fim para
B2) **se** $|B| \times |\langle t, f, d \rangle| \geq M$ **então**
B2.1) *Ordene*(B) /* $t \uparrow, f \downarrow, d \uparrow$ */
B2.2) *Comprima*(B)
B2.3) *Grave*(B, F')
$B \leftarrow \emptyset$
$R \leftarrow R + 1$
fim se

Fase C: $F \leftarrow$ *Intercale*(F', R)

Algoritmo 2.1: Geração seqüencial de arquivos invertidos ordenados por freqüência.

Modelo matemático. A Equação 2.1 apresenta a expressão matemática para o tempo de execução da indexação seqüencial, considerando o Algoritmo 2.1 e as variáveis do modelo matemático introduzido nas Tabelas 1.1 e 1.2. Nessa equação, o valor R, o número de *runs* produzidos na indexação, é derivado de outros parâmetros, a saber, o número de pontos de indexação x, o tamanho em bytes de cada ponto de indexação, $|\langle t, f, d \rangle|$, e o tamanho em bytes da memória disponível, M, de forma que $R = x|\langle t, f, d \rangle|/M$.

O tempo da fase A corresponde à leitura da coleção, ao *parsing* de suas palavras, ordenação do vocabulário e construção da função *hash* perfeita. As constantes associadas ao tempo de construção da função não podem ser determinadas, pois, como mostrado na Seção 2.2.2, o algoritmo responsável por esse cálculo é aleatório. Pode-se, entretanto, provar que esse tempo é $O(v)$, se v é o número de chaves para as quais a função é construída.

$$
\begin{aligned}
t_{SEQ} &= t_{SEQ_A} + t_{SEQ_B} + t_{SEQ_C} \\
\\
t_{SEQ_A} &= t_{SEQ_{A1.1}} + t_{SEQ_{A1.2}} + t_{SEQ_{A2}} + t_{SEQ_{A3}} \\
t_{SEQ_A} &= n(t_r + t_p) + (1.2v\log v)|w_V|t_{cs} + O(v) \\
\\
t_{SEQ_B} &= t_{SEQ_{B1.1}} + t_{SEQ_{B1.2}} + t_{SEQ_{B2.1}} + t_{SEQ_{B2.2}} + t_{SEQ_{B2.3}} \\
t_{SEQ_{B2.1}} &= R\left(\lambda\tfrac{x}{R}\right)t_{ct} = \lambda x t_{ct} \\
t_{SEQ_B} &= n(t_r + t'_p) + \lambda x t_{ct} + f'(t_z + t_r) \\
\\
t_{SEQ_C} &= f'\left(\tfrac{R-1}{RM}t_s + t_r + t'_z\right) + x\lceil\log R\rceil t_{ct} + f(t_r + t_z)
\end{aligned}
\tag{2.1}
$$

A fase B inclui uma nova leitura do texto e um segundo *parsing*, diferente do primeiro, pois a estrutura de dicionário utilizada é outra e agora devem-se contabilizar todas as ocorrências dos termos, montando as listas invertidas. Faz também parte desta fase a ordenação das triplas, para a qual desenvolveu-se o método linear descrito na Seção 4.4.2. Utiliza-se a constante de proporcionalidade λ para se representar o tempo tomado pela ordenação. Contabiliza-se também a compressão das triplas e sua gravação do arquivo temporário. O tamanho deste último é dado em bytes por f', valor maior que o tamanho f do arquivo final, já que o primeiro contém triplas $\langle t, f, d\rangle$, enquanto que o outro é composto apenas por duplas $\langle f, d\rangle$.

Na fase C, todo o arquivo temporário deve ser lido e descomprimido, daí os fatores $f't_r$ e $f't'_z$, respectivamente. As triplas encontram-se divididas em R *runs* e são carregadas para a memória em R listas de blocos de dados. Os blocos ocuparão toda a memória M disponível, onde permanecerão comprimidos e fornecerão dados para o *heap*. Quando esgotam os dados em memória de uma lista r, os próximos blocos do *run* r devem ser lidos e substituir a lista esgotada, o que ocorrerá aproximadamente f'/M vezes.

Assumindo independência na ordem com que esgotam os dados em memória, sempre que uma nova lista é lida do disco, existe uma probabilidade $1/R$ do cursor de leitura do arquivo temporário estar exatamente sobre o *run* a ser lido, o que se dá caso esse *run* tenha sido o último carregado para a memória. Portanto, a cada nova leitura de uma lista de blocos, a cabeça de leitura do disco será reposicionada com probabilidade $(R-1)/R$. Daí o fator $f'\frac{R-1}{RM}t_s$ no tempo de execução, destacando o ganho proporcionado por essa intercalação em relação à proposta em [85]. Nesta referência, o tempo gasto em *seeks* é dado por $f't_s/b$, onde b é o tamanho de um

bloco de disco (tipicamente 4 Kbytes), o que resulta em valores bastante altos à medida que o texto cresce. Além disso, o tempo t_{SEQ_C} contabiliza $x\lceil \log R \rceil$ comparações de triplas no *heap* e o tempo de gravação do arquivo invertido final, também comprimido.

2.2 Acesso ao Vocabulário

Uma questão central no projeto de algoritmos de indexação é como armazenar e acessar de forma eficiente o vocabulário do texto. A cada palavra encontrada no texto, a estrutura contendo o vocabulário é acessada e, caso essa operação não seja feita de forma eficiente, o desempenho do algoritmo é comprometido. Na linguagem de projeto de algoritmos, a estrutura de dados que implementa operações de inserção, busca e remoção de itens em um conjunto dinâmico é chamada **dicionário**. Nesta seção, discutem-se alternativas para a implementação eficiente de um dicionário para o armazenamento dos termos pertencentes ao vocabulário de um texto.

Quando ainda não se conhece o conjunto de itens a serem armazenados no vocabulário, a estrutura que apresenta melhor desempenho é a tabela *hash* tradicional, que oferece um caso médio de $O(1)$ para inserções e acessos a itens. Porém, essa tabela está sujeita ao fenômeno das colisões, o que leva a um pior caso de $O(n)$, se n é o número de itens presentes. Por outro lado, quando já se conhece o conjunto de itens, uma diferente tabela *hash* pode ser pré-computada, permitindo acessos em tempo $O(1)$, sem o risco de colisões. Devido a essa e outras propriedades favoráveis, essa nova tabela *hash* é dita perfeita. Ambas implementações da estrutura de dicionário delineadas acima serão discutidas em detalhes nas seções a seguir.

2.2.1 Tabelas *hash*

Uma **tabela *hash*** é uma estrutura de dados eficiente para a implementação de um dicionário. Para se armazenar um conjunto de itens pertencentes a um universo U em uma tabela T, uma alternativa óbvia é alocar para T um tamanho $|U|$, de forma que cada item seja armazenado diretamente em sua posição correspondente. Entretanto, caso U seja muito grande, isso pode ser impossível, dada a memória disponível em um computador típico. Para cadeias com até $|w_{max}|$ caracteres e um alfabeto de σ elementos, o valor de $|U|$ é

$$\sum_{i=1}^{|w_{max}|} \sigma^i = \frac{\sigma^{|w_{max}|+1} - 1}{\sigma - 1} - 1.$$

Para $|w_{max}| = 256$ e $\sigma = 36$ (26 letras e 10 dígitos), os valores correspondentes às convenções adotadas neste trabalho, tem-se $|U| \approx 10^{400}$, um valor muito superior à capacidade de qualquer computador existente.

Normalmente, porém, o conjunto $N \subset U$ de chaves a serem armazenadas em um dicionário é muito inferior ao universo U. Se $n = |N|$ é o número de chaves consideradas, então uma tabela

hash é uma estrutura com requisitos de armazenamento da ordem de $\Theta(n)$ e que, ainda assim, permite acesso em tempo $O(1)$, isto é, constante. Isso é possível através do uso de uma **função *hash***, que mapeia as chaves $k_i \in N, 0 \leq i < n$ em posições $0 \leq h(k_i) < m$ da tabela.

O valor $m \geq n$ é o tamanho da tabela e a razão $\alpha = n/m$ é um valor menor ou igual a 1, chamado **fator de carga** da tabela, que dá uma idéia da densidade de distribuição das chaves. Quanto menor o valor de α, menos provável é que duas chaves distintas k_i e k_j colidam, apresentando um mesmo mapeamento $h(k_i) = h(k_j)$. Por menor que seja α, colisões são quase impossíveis de serem evitadas, uma vez que, à medida que se consideram mais chaves, a probabilidade de que o espalhamento sempre funcione cai muito rapidamente.

Além do fator de carga, outro aspecto fundamental para permitir um bom espalhamento das chaves e, logo, um funcionamento eficiente da tabela, é a escolha de uma função *hash* apropriada. Se as chaves k_i são números aleatórios, uma boa escolha é tomar $h(k_i) = k_i \bmod m$. Essa função explora a aleatoriedade dos bits inferiores das chaves e permite um bom espalhamento das mesmas no intervalo $[0, m-1]$. Em geral, uma função *hash* será adequada se satisfizer a suposição do espalhamento uniforme [42, 16]: cada chave pode ser mapeada para qualquer uma das m posições da tabela com igual probabilidade.

Ainda assim, uma entrada de dados maliciosa ou tendenciosa pode levar ao pior caso de acesso a uma tabela *hash*, de tempo $\Theta(n)$; qualquer função *hash* fixa está sujeita a esse comportamento. A solução é escolher uma função aleatoriamente, tornando-a independente das chaves que serão armazenadas na tabela. Essa abordagem, chamada ***hashing* universal** [42, 16], leva a um bom desempenho no caso médio, não importando como as chaves sejam escolhidas. Uma classe universal de funções *hash* pode ser definida da seguinte forma: toma-se o tamanho m da tabela como um número primo, particiona-se cada chave k em r partes $\langle k[1], k[2], \ldots, k[r] \rangle$ e constrói-se um vetor contendo valores aleatórios $\langle a[1], a[2], \ldots, a[r] \rangle$. Cada $k[i]$ e $a[i]$ deve ser menor que m. Uma função *hash* universal h_a pode então ser definida como

$$h_a(k) = \left(\sum_{i=1}^{r} k[i] \times a[i] \right) \bmod m \tag{2.2}$$

Entretanto, o objetivo da estrutura de dados dicionário neste trabalho é armazenar cadeias de caracteres (correspondentes aos termos do vocabulário) e não valores inteiros. A mesma teoria desenvolvida para chaves numéricas pode ser aplicada convertendo-se as cadeias de caracteres em inteiros. Nesse ponto, a teoria de *hashing* universal ajusta-se perfeitamente, pois as cadeias podem ser naturalmente particionadas pelos caracteres que as compõem. Assim, uma cadeia k composta pelos caracteres $k[1]k[2]\ldots k[r]$ pode ter sua função *hash* dada pela Equação 2.2.

A real implementação da função *hash* evita a multiplicação de cada caractere $k[i]$ pelo elemento $a[i]$ correspondente no vetor aleatório, pré-computando os valores, assim como descrito na Seção 4.3.1. Os resultados experimentais mostram que esse esquema proporciona um número muito pequeno de colisões, caso o fator de carga α da tabela seja mantido não muito

alto (por exemplo, entre 0.25 e 0.75), como mostra a Figura 5.5.

Outra alternativa no projeto de tabelas *hash* é como resolver colisões de chaves. Duas alternativas populares são o **encadeamento** e o **endereçamento aberto**. A primeira procede construindo, em cada posição da tabela onde chaves colidem, uma lista encadeada contendo essas chaves. A segunda resolve colisões escolhendo, no corpo da própria tabela, uma nova posição para armazenar uma chave que colidiu. Esta última alternativa proporciona melhor tempo de execução e economia de espaço, desde que a escolha da nova posição seja bem feita.

Uma possibilidade simples para a escolha da nova posição em endereçamento aberto é percorrer linearmente a tabela a partir do ponto onde deu-se a colisão, armazenando a chave na primeira posição livre. Essa abordagem é ingênua e sofre de um problema chamado *clustering*: à medida que a tabela é preenchida, formam-se longos aglomerados de chaves que colidiram, o que degrada o desempenho da estrutura. Dentre outras formas de escolha dessa nova posição, a mais atraente é o ***hashing* duplo**, pelo qual uma segunda função *hash* é calculada. Essa segunda função deve ser simples de se computar e é usada para percorrer a tabela até se encontrar uma posição livre para a chave. *Hashing* duplo é a estratégia utilizada neste trabalho e a implementação da busca de uma chave na tabela *hash* é descrita em detalhe pelo Algoritmo 4.1.

2.2.2 *Hashing* perfeito

Se, de alguma forma, uma função *hash* h for tal que, para todo k_i e k_j pertencentes a N, tem-se $h(k_i) = h(k_j)$ se e somente se $i = j$, então h não provoca colisões entre chaves e é dita uma função *hash* **perfeita**. Se h, além de perfeita, mapeia as chaves em um intervalo de tamanho $m = n$, então há uma correspondência bijetiva entre as chaves e as posições da tabela e o fator de carga é $\alpha = 1$. Nesse caso, h é então uma função *hash* perfeita **mínima**. Esse tipo de função garante que o acesso à tabela requer apenas um cálculo do endereço da chave e que não há posições vazias. Finalmente, se uma função *hash* tem a propriedade que, se $x_i < x_j$, então $h(x_i) < h(x_j)$, ela é dita **preservadora da ordem**.

Uma função *hash* perfeita mínima preservadora da ordem (ou OPMPHF, de *Order Preserving Minimal Perfect Hash Function*) permite a localização das chaves em tempo constante sem sobrecarga de espaço e pode ainda aproveitar a propriedade da ordenação das chaves. Obviamente, uma OPMPHF não pode ser uma função genérica, mas sim específica para um conjunto N de chaves, pois ela é uma função de busca pré-calculada. Ela é adequada somente para casos em que esse cálculo é possível (dado que o conjunto de chaves é previamente conhecido), possibilitando economias de espaço e facilidade de programação. Há várias técnicas para a construção de OPMPHFs, como [23, 24]. Um método particularmente elegante, baseado em grafos aleatórios, é apresentado em [20, 36] e consiste na técnica utilizada neste trabalho.

Como citado na página 20, o uso de uma OPMPHF pode ser útil na construção seqüencial de arquivos invertidos e é essencial nas abordagens distribuídas adotadas neste trabalho. Esse tipo de função permite a representação dos termos do vocabulário por inteiros, eliminando a

necessidade do armazenamento de *string*. As operações de acesso ao vocabulário podem ser realizadas em tempo constante, sem o risco de colisões ou comparações entre *strings* e os custos em espaço da estrutura dicionário são reduzidos. Com isso, mais memória torna-se disponível para o *buffer* de triplas durante a indexação, o que pode diminuir o número de *runs* e, com isso, o número de *seeks* realizados na fase de intercalação, levando até mesmo à redução do custo desta fase.

A construção de uma OPMPHF assume, além do conhecimento prévio das n chaves, a existência de duas (ou mais, como mostra-se a seguir) funções *hash* normais $h_j(k)$ que mapeiem as chaves em um intervalo $[0, m-1]$, assim como a função mostrada na Equação 2.2. Têm-se, portanto,

$$h_a(k) = \left(\sum_{i=1}^{|k|} k[i] \times a[i] \right) \bmod m \qquad \text{e} \qquad h_b(k) = \left(\sum_{i=1}^{|k|} k[i] \times b[i] \right) \bmod m,$$

onde $k[i]$ é o i-ésimo caractere da cadeia k, de tamanho $|k|$; $a[i]$ e $b[i]$ são pesos aleatórios distintos, que formam duas diferentes funções *hash* universais h_a e h_b.

A OPMPHF depende, além desses itens, de um arranjo g bastante especial, capaz de mapear os números $0, \ldots, m-1$ no intervalo $[0, n-1]$. A avaliação da OPMPHF é dada então por

$$h(k) = [g(h_a(k)) + g(h_b(k))] \bmod n. \tag{2.3}$$

A principal questão é como encontrar uma função g adequada. A abordagem aqui adotada encontra-se proposta em uma série de trabalhos por Czech et al. [20, 36, 47, 19] e baseia-se em grafos e hipergrafos aleatórios. Ela é descrita de forma simplificada a seguir.

A tarefa de encontrar um mapeamento g pode ser visualizada em um grafo com vértices rotulados $0, \ldots, m-1$ e arestas definidas por pares $\langle h_a(k_i), h_b(k_i) \rangle$, para cada uma das chaves $0 \leq k \leq n$. Assim, a cada chave corresponde uma aresta do grafo e os valores das funções *hash* h_a e h_b definem sobre quais vértices a aresta incidirá. Então, se cada aresta for rotulada por $h(k_i)$ – o valor desejado para a função *hash* perfeita aplicada sobre k_i – os rótulos dos vértices do grafos consistirão na função g procurada. O valor de $h(k_i)$ pode ser definido arbitrariamente; a escolha mais natural, feita neste trabalho, é ordenar as chaves $k_1, \ldots, k_n$ lexicograficamente e definir $h(k_i) = i$, a posição da i-ésima chave no conjunto ordenado.

O problema transforma-se agora em encontrar um mapeamento g de vértices de um grafo em inteiros $0, \ldots, n-1$ tal que, para cada aresta $\langle h_a(k_i), h_b(k_i) \rangle$, o mapeamento resulta em $[g(h_a(k_i)) + g(h_b(k_i))] \bmod n = h(k_i)$, o rótulo da mesma aresta. Para grafos quaisquer, tal mapeamento pode ser difícil de se encontrar, mas para grafos acíclicos[2], uma função como g pode ser facilmente calculada.

[2]Um grafo aciclico é aquele que, percorrendo cada uma de suas arestas uma única vez, é impossível retornar a um vértice já visitado, isto é, não há ciclos.

Um vértice qualquer v ainda não processado é escolhido e define-se $g(v) = 0$. As arestas incidentes sobre esse vértice são então percorridas e os vértices alcançados através de cada aresta são rotulados com o mesmo rótulo da aresta utilizada, isto é, a função h para a chave correspondente à aresta. A partir daí percorrem-se os vértices que podem ser alcançados a partir daqueles que acabaram de ser rotulados. Os vértices desse novo nível são, por sua vez, rotulados com a diferença entre os rótulos da aresta usada para alcançá-los e do vértice a partir do qual eles foram alcançados. Quando todos os vértices que podem ser atingidos a partir de v foram percorridos, verifica-se se ainda há vértices não processados no grafo. Em caso afirmativo, o processo continua até que todos os vértices tenham sido rotulados, quando então o mapeamento g está completo.

Se, em algum momento desse processo, alcança-se um vértice já rotulado por um valor diferente daquele que se procura agora atribuir, o grafo é cíclico e não permite a construção do mapeamento g. Assim, essa técnica de rotulação do grafo possibilita a detecção automática de ciclos. Ela é também eficiente, pois executa em tempo $O(m+n)$, proporcional ao número de vértices e arestas no grafo. O Algoritmo 2.2 formaliza o processo de geração de uma função *hash* perfeita pela técnica descrita. Nota-se a definição do valor de m como $\alpha^{-1}n$, onde α é o fator de carga definido na Seção 2.2.1. A escolha de α neste caso não é livre, mas possui restrições, como discutido a seguir. As funções para rotulação do grafo como um todo e de um conjunto de arestas são apresentadas nos Algoritmos 2.3 e 2.4, respectivamente.

Resta ainda determinar a probabilidade com a qual um grafo acíclico é produzido pela técnica descrita. Caso somente grafos cíclicos sejam produzidos, nunca se obterá o mapeamento desejado e os algoritmos mostrados não têm utilidade alguma. Essa é uma questão bastante interessante, relacionada à teoria de grafos aleatórios e tem uma íntima ligação com a escolha do número de vértices m do grafo. Quanto maior o valor de m, mais esparsas são as arestas do grafo e maior a probabilidade de que ele seja acíclico. Análises baseadas na teoria de grafos aleatórios mostram que, para $m \leq 2n$, a probabilidade de gerar um grafo aleatório tende a zero à medida que n cresce, pois o grafo torna-se muito denso e é difícil evitar o surgimento de ciclos. Por outro lado, quando $m > 2n$, a probabilidade de que um grafo gerado seja acíclico é

$$\sqrt{\frac{m-2n}{m}}.$$

Tomando, por exemplo, $m = 2.5n$ (isto é, $\alpha^{-1} = 2.5$) e considerando que a tabela de mapeamento g contém inteiros de 4 bytes, os requisitos de espaço da função *hash* perfeito são de $2.5 \times 4 = 10$ bytes por chave. No caso de chaves constituídas por cadeias de caracteres, essa demanda é menor que o custo de se armazenarem as próprias cadeias em memória, mas ainda assim pode ser alta. Uma alternativa para se reduzir o valor de α^{-1} é não se utilizar mais grafos tradicionais, mas sim hipergrafos (ou r-grafos), nos quais cada aresta conecta um número qualquer r vértices e não mais apenas 2. Empregando-se trigrafos ($r = 3$), pode-se utilizar um valor

Entradas: N, o conjunto de n chaves para as quais a função *hash* será criada
Saídas: g, a e b, mapeamentos que permitirão o cálculo da função *hash* pela Equação 2.3

```
m ← α^{-1}n
repita
  para i ← 1 até max |k| faça
    a[i] ← ⟨valor aleatório⟩
    b[i] ← ⟨valor aleatório⟩
  fim para
  Crie um grafo G = (V,E), com V = {0,...,m−1} e E = {⟨h_a(k_i), h_b(k_i)⟩ ∀k_i ∈ N}
  RotuleGrafo(V)
até rotulação com sucesso (G é acíclico)
```

Algoritmo 2.2: Processo de geração de uma função *hash* perfeita.

Entradas: V, o conjunto de vértices do grafo
Saídas: g, o mapeamento a ser gerado para produzir a função *hash* h

```
para v ∈ V faça
  g[i] ← indeterminado
fim para
para v ∈ V faça
  se g[i] = indeterminado então
    RotuleVértice(v, 0)
  fim se
fim para
```

Algoritmo 2.3: Algoritmo para rotular um grafo, determinando o mapeamento g.

Entradas: v, o vértice a partir do qual realiza-se o mapeamento; c, o rótulo a ser aplicado a v
Saídas: g, o mapeamento modificado pela rotulação de v e seus adjacentes

```
se g[v] ≠ indeterminado então
  se g[v] ≠ c então
    retorne falha: o grafo é cíclico
  senão
    retorne sucesso: este vértice já foi visitado
  fim se
fim se
g[v] ← c
para u ∈ adjacentes(v) faça
  RotuleVértice(u, h(⟨v,u⟩) − g[v])
fim para
```

Algoritmo 2.4: Algoritmo para rotular um vértice e seus adjacentes.

de α^{-1} tão baixo quanto 1.23. Um novo vetor de valores aleatórios c deve ser gerado e a função *hash* passa a ser definida como

$$h(k) = [g(h_a(k)) + g(h_b(k)) + g(h_c(k))] \bmod n. \tag{2.4}$$

Apesar de reduzir os custos de espaço da função *hash* perfeita, o uso de trigrafos aumenta o tempo de acesso ao dicionário, pois requer o cômputo de mais uma função *hash* auxiliar h_c. Além disso, o processo de rotulação do grafo não pode mais ser feito concorrentemente com a detecção de ciclos como descrito anteriormente. Ciclos devem ser detectados previamente à rotulação, utilizando a seguinte propriedade de r-grafos:

> *Um r-grafo é acíclico se e somente se a remoção repetida de arestas contendo apenas vértices de grau 1 (isto é, vértices sobre os quais incide apenas uma aresta) elimina todas as arestas do grafo.*

Neste trabalho implementa-se esta última abordagem para a busca do mapeamento g, permitindo-se a geração tanto de digrafos como de trigrafos. Detalhes desta implementação são apresentados na Seção 4.3.3.

2.3 Processamento de Consultas

Uma vez que já foi descrita a estrutura de um arquivo invertido e como ele pode ser construído, armazenado e acessado eficientemente, discute-se agora como utilizá-lo para localizar, de maneira eficaz, a informação dentro do texto indexado. Serão examinados dois tipos de consultas:

(i) *lógicas*, nas quais os termos são conectados por expressões como "E" e "OU";

(ii) *ordenadas*, nas quais o próprio sistema de recuperação procura classificar as respostas de acordo com sua suposta relevância, apresentando-as na ordem esperada de importância.

Primeiramente, seguem-se alguns conceitos centrais em relação à eficácia de um sistema de recuperação de informação. Um documento é dito **relevante** se, após examiná-lo, o usuário do sistema indica que ele satisfaz sua necessidade de informação; logo, somente o usuário pode avaliar a relevância de um documento, cabendo ao sistema apenas utilizar heurísticas para tentar prevê-la.

Uma resposta a uma consulta possui alta **precisão** se, dentre os documentos apontados, grande parte é relevante. Por outro lado, uma resposta apresenta alta **revocação** se contém grande parte dos documentos relevantes existentes no banco de dados. Precisão e revocação são duas características desejáveis, mas em geral conflitantes: incluindo mais documentos na resposta, aumenta-se a revocação, porém documentos não relevantes acabam por ser selecionados, o que diminui a precisão. Na direção inversa, restringindo-se os documentos incluídos na

resposta, aumenta-se a precisão, mas documentos relevantes podem ser excluídos, prejudicando a revocação.

2.3.1 Consultas lógicas

Uma **consulta lógica** (ou Booleana) consiste em uma lista de termos combinados pelos conectores lógicos "E", "OU" e "NÃO". As respostas à consulta são aqueles documentos que satisfazem exatamente a condição estipulada pelos termos. Uma possível consulta sobre o banco de dados da Tabela 2.1 seria "ainda E onda" e retornaria os documentos 4 e 5, aqueles que contêm ambos os termos. Já a consulta "ainda OU onda" retornaria os documentos 1, 3, 4, 5, 6, e 9, aqueles que contém qualquer um dos termos.

Consultas lógicas são de formulação bastante simples e podem apresentar bons resultados, sobretudo para usuários que conhecem bem os documentos do banco de dados a ser pesquisado. Além disso, o formalismo por trás do modelo – a lógica Booleana – é bastante conhecido e possui várias propriedades interessantes e úteis para a pesquisa.

Entretanto, a determinação de respostas pode variar muito dependendo de como são escolhidos os conectivos para a consulta. Enquanto que a combinação dos termos através de operadores "E" pode restringir demais os resultados (prejudicando a revocação), o uso do operador "OU" pode incluir muitos documentos não relevantes (afetando a precisão). Além disso, o modelo não é capaz de apontar uma ordem de relevância entre os documentos da resposta, cabendo ao usuário determinar manualmente quais satisfazem mais sua necessidade de informação. Vários trabalhos foram feitos na tentativa de se atenuar esse casamento exato (alguns utilizando lógica difusa [64, 10]) e de se introduzir uma ordenação entre as respostas [9, 41].

2.3.2 Ordenação das respostas

Uma possível solução para as deficiências do modelo lógico é realizar uma **consulta ordenada**. O usuário informa um conjunto de termos (sem conectores), a partir dos quais o sistema de recuperação emprega uma heurística para determinar uma medida de similaridade entre essa consulta e os documentos do banco de dados. Baseado nesse indicador numérico, o sistema aponta os r documentos que mais se assemelham à consulta formulada e os retorna ao usuário.

Muita pesquisa tem sido realizada no esforço de se encontrar uma medida de similaridade e outras técnicas de ordenação que possam manter precisão e revocação razoavelmente altas. Entre as técnicas que produzem os melhores resultados na determinação da similaridade está a **medida do cosseno**, utilizada no modelo vetorial [67, 66]. Nesse arcabouço, são levados em consideração um peso do termo em relação ao documento (por exemplo, sua freqüência no documento), a freqüência do termo na coleção (termos mais raros têm maior importância) e o tamanho do documento (documentos mais curtos são privilegiados). Assim, consulta e documentos são representados por vetores em um espaço ortonormal onde a base são os termos

da coleção e similaridade é determinada pelo cosseno entre os vetores.

Dentre as propriedades interessantes dessa abordagem, observa-se que a pesagem de termos melhora o desempenho do mecanismo, ao contrário do modelo lógico, que trata todos os termos de forma homogênea. A determinação de um grau de similaridade comporta-se de forma mais suave que um casamento exato e permite a ordenação dos documentos, elevando documentos mais relevantes às primeiras posições da resposta.

Um aspecto bastante criticado no modelo vetorial é o fato de se assumir uma independência entre os termos do vocabulário, que formam os vetores-base do espaço ortonormal onde se representam documentos e consultas. Isso pode levar à desconsideração de interrelações importantes entre os termos [60]. Outros modelos que utilizam medidas de similaridade e pesagem de termos para ordenar respostas a consultas são o modelo vetorial generalizado [86] (que procura considerar as interrelações entre termos) e os modelos probabilísticos, incluindo as redes de inferência [56, 82, 43].

Além de serem capazes de retornar os documentos que mais satisfaçam aos critérios estabelecidos pela consulta, os modelos também devem cuidar para que o processamento seja realizado de forma eficiente, de forma que a resposta possa ser apresentada rapidamente ao usuário. Para consultas lógicas **conjuntivas** (nas quais os termos são conectados pelo operador "E"), a técnica para se reduzir ao mínimo tanto o consumo de memória quanto o tempo de processamento é avaliar os termos em ordem crescente de freqüência na coleção. Assim, a lista de documentos candidatos à resposta é inicializada com a menor lista de todos os termos e, como a cada novo termo processado é realizada uma interseção, garante-se que o número de candidatos nunca cresce. Consultas não conjuntivas podem ser processadas de forma semelhante convertendo-as para essa forma.

Já para consultas ordenadas, o problema é mais complexo, pois a classificação de cada documento não é apenas um "SIM" ou um "NÃO", mas uma medida de similaridade. A abordagem tradicional é, para cada documento, manter um **acumulador** contendo sua similaridade com a consulta. Cada termo da consulta é processado e os acumuladores correspondentes aos documentos em sua lista invertida são incrementados com a contribuição daquele termo para a similaridade entre documento e consulta. Entretanto, manter um acumulador para cada documento da coleção pode ser um custo muito alto. Uma alternativa é alocar acumuladores apenas sob demanda, isto é, somente se um documento aparecer na lista invertida de algum termo da consulta, mas ainda assim os gastos com memória podem ser demasiados.

Abordagens mais sofisticadas começaram a ser discutidas em [32]. Duas heurísticas para se reduzir a memória e o tempo de processamento na avaliação de consultas ordenadas são apresentadas em [53]. Uma delas, chamada "*STOP*", interrompe o processamento quando o número de acumuladores atinge um determinado limite. A outra, chamada "*CONTINUE*", continua quando esse limite é alcançado, porém novos acumuladores não são mais acrescentados, somente os já existentes são atualizados. A primeira reduz significativamente gastos com memória e

tempo de processamento, mas prejudica a qualidade da resposta. Já a segunda reduz o consumo de memória, mas não leva a ganhos tão grandes no tempo de processamento; entretanto, suas respostas têm qualidade comparável ao método tradicional de avaliação.

Uma técnica que leva a grandes reduções nos custos de avaliação de consultas é apresentada por Persin em [57, 58]. Ela permite grandes cortes na memória consumida e no tempo de processamento através do reconhecimento prematuro dos documentos que terão chances de fazer parte da resposta. Por exemplo, documentos contendo apenas uma ocorrência de um termo da consulta dificilmente entrarão no topo da ordem dos relevantes e podem ser descartados; assim, bastante tempo é economizado evitando-se ler listas invertidas completas. A eficácia da recuperação (precisão e revocação) não é afetada pela heurística, que elimina apenas documentos situados nas posições inferiores da resposta. Entretanto, para ser eficiente, essa técnica requer que as listas invertidas tenham os pares $\langle d, f \rangle$ ordenados pelas freqüências f e não mais pelos documentos d. Essa modificação tem impacto sobre os algoritmos de construção de arquivos invertidos, porém não prejudica a compressão das listas, que pode até apresentar taxas superiores à de índices ordenados por documentos.

Neste trabalho são gerados índices com as modificações propostas em [58]. Porém, não se implementa nenhum mecanismo para processamento de consultas (seqüencial ou distribuído), ficando isso a cargo do usuário dos índices construídos como produto desta dissertação.

Capítulo 3

Algoritmos Distribuídos

Este capítulo descreve a principal contribuição deste trabalho: o projeto de uma família de algoritmos distribuídos para a indexação de grandes volumes de texto. Como citado na Introdução, o uso de *hardware* paralelo ou distribuído é uma alternativa conhecida para lidar com as crescentes demandas dos sistemas de informação modernos. Entretanto, os trabalhos envolvendo esse tipo de arquitetura concentram-se quase que totalmente na questão do processamento de consultas [14, 13, 81, 63], deixando de lado a preocupação com uma indexação eficiente.

Stanfill investigou algoritmos para indexação em arquiteturas paralelas, mais especificamente, na CM-2 da *Thinking Machines*, uma máquina SIMD. Entretanto, tal trabalho, mesmo que aplicável a outras arquiteturas paralelas além da CM-2, restringe-se ao emprego em máquinas SIMD. Em respeito a arquiteturas distribuídas, os únicos trabalhos conhecidos sobre indexação são [61, 68]. Entretanto, o algoritmo apresentado nesses trabalhos utiliza somente a memória principal das máquinas para o armazenamento das ocorrências, o que limita sua aplicabilidade para grandes coleções de texto.

O primeiro trabalho considerando o uso de memória secundária no processo distribuído de indexação é [62], no qual está baseada esta dissertação. Nele é descrita uma família de três algoritmos distribuídos para a indexação de textos, juntamente com a caracterização de seu ambiente de execução e resultados analíticos. Os algoritmos diferem entre si na maneira como cada máquina trata os dados que não serão armazenados localmente. Nas seções a seguir, apresenta-se com maior detalhe o ambiente considerado para a execução dessa família de algoritmos e então descreve-se em detalhes cada uma das três abordagens para indexação distribuída.

3.1 Arquitetura do Ambiente

Neste trabalho, assim como em outros da literatura [80, 81, 63, 62], assume-se um ambiente de execução formado por uma rede de alta velocidade conectando um número de estações de trabalho, uma arquitetura chamada *shared-nothing memory* [80]. Idealmente, as estações estariam conectadas por uma chave (*switch*) rápida, que lhes permitiria comunicar sem contenção.

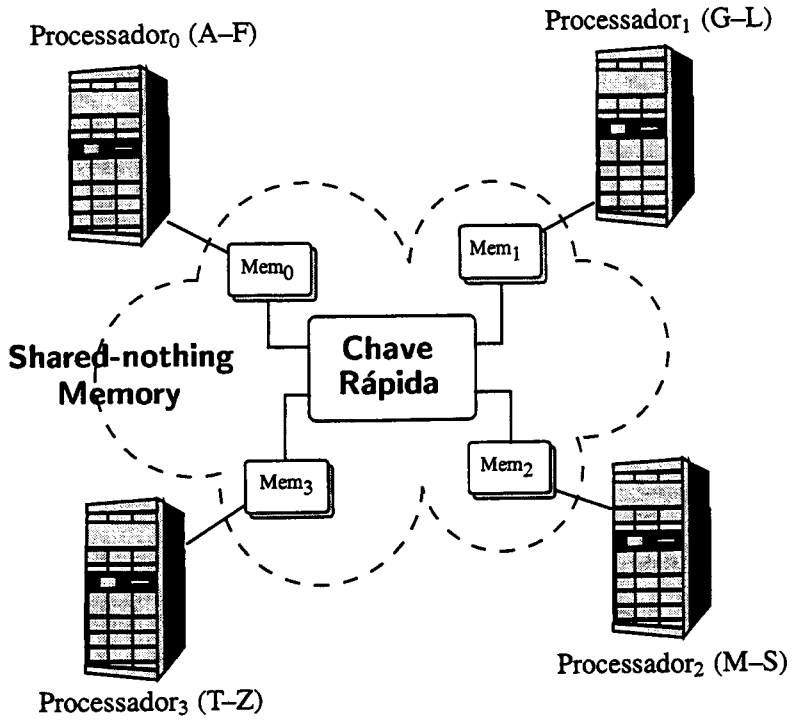


Figura 3.1: Ambiente de execução dos algoritmos distribuídos.

A Figura 3.1 ilustra a arquitetura considerada. Na mesma figura, mostra-se como o arquivo invertido é distribuído entre os processadores: cada um armazena as listas completas para um subconjunto do vocabulário, uma organização chamada índice global, descrita na Seção 3.1.2 a seguir. No exemplo, termos iniciados por A até F ficam no processador 0, termos de G a L no processador 1 e assim em diante. Na realidade, a implementação do particionamento procura destinar um número igual de termos a cada máquina, como descrito na Seção 3.2.

3.1.1 Considerações sobre o ambiente de execução

Discutem-se agora algumas suposições feitas sobre o ambiente de execução dos algoritmos distribuídos desenvolvidos neste trabalho e sua influência nos resultados obtidos. Cumpre adiantar que nenhuma dessas considerações inviabiliza ou reduz a aplicabilidade da abordagem aqui proposta; a maioria delas somente formaliza o estado da tecnologia atual, algumas são feitas para permitir a comparação dos algoritmos, outras apenas possibilitam a simplificação das análises ou da implementação.

Uma das considerações centrais para este trabalho é que as máquinas executando os algoritmos distribuídos estejam conectadas por uma rede rápida, pela qual elas possam se comunicar livremente, em qualquer ordem, sem contenção. Uma situação ideal (como mostrado na Figura 3.1) é a presença de uma chave ou *switch*, um dispositivo que assegura, via hardware, a independência na comunicação entre dois pares de máquinas. Entretanto, os algoritmos

desenvolvidos são aplicáveis a redes apresentando taxas nominais hoje comuns (como 10 e 100 Mbps) mesmo que elas não sejam providas de *switches*, mas desde que elas estejam dedicadas à execução dos algoritmos. Em geral, se a rede não for muito mais lenta que os discos das máquinas, situação que acontece hoje vem se acentuando cada vez mais, as suposições feitas são válidas. Vale salientar que, em uma situação contrária, com redes muito lentas ou carregadas, os algoritmos ainda são úteis, porém algumas conclusões relativas a sua eficiência podem mudar, como será destacado na Seção 5.4.2.

As análises também supõem que as máquinas utilizadas nos algoritmos distribuídos possuem o mesmo poder de processamento, seja em termos de capacidade da memória principal e dos discos, ou em relação à velocidade do processador, placa de rede, memória e controladora de disco. Essa suposição simplifica as expressões das análises de tempo de execução e não é de aplicação restrita, sendo seguida pelas máquinas utilizadas nos experimentos deste trabalho, compradas de um mesmo fornecedor e com a mesma configuração. Para propósitos de simplificação da análise, também supõe-se que a coleção encontra-se igualmente distribuída entre as máquinas, armazenada nos discos locais, antes do início da execução. Dessa forma, parâmetros como o tamanho da coleção em bytes n, o número de documentos c, o número de pontos de indexação x e o número v de termos no vocabulário são supostos serem idênticos entre as diversas máquinas.

Outra suposição, já assumida para o algoritmo seqüencial e que continua válida para a família distribuída, é que a memória principal de cada máquina não é capaz de comportar o índice para o texto em seu disco. Essa é a principal diferença conceitual da abordagem deste trabalho para aquela apresentada em [61, 68], pois permite o emprego dos algoritmos aqui descritos para coleções de qualquer tamanho. É também essa suposição, juntamente com a afirmação de que a velocidade da rede é maior ou igual à dos discos, que permite a variedade de abordagens aqui propostas. Caso cada máquina pudesse armazenar todo seu índice em memória ou caso a rede fosse muito lenta, a alternativa óbvia seria construir os índices localmente e somente depois trocar as listas e formar o índice global, como proposto pelo algoritmo LL descrito na Seção 3.3.

Ainda outra suposição que também era válida para o algoritmo seqüencial é que os processadores das máquinas são rápidos o suficiente para que o uso de compressão dos índices leve a ganhos de tempo de execução dos algoritmos. Realmente, isso é válido para os processadores atuais, como já foi aludido na teoria por [51] e é mostrado experimentalmente na Seção 5.3.1. Tal fenômeno ocorre porque, devido à grande redução dos custos em espaço proporcionada pela compressão dos índices, o uso de recursos como disco e rede cai o suficiente para mais que compensar o tempo de CPU gasto comprimindo-se os índices. A tendência ao longo do tempo é esse fato tornar-se ainda mais acentuado, já que a velocidade dos processadores cresce em uma proporção maior que a de dipositivos periféricos, como os discos.

Uma consideração feita na implementação dos algoritmos é que tanto as máquinas como a

rede são de alta confiabilidade e pouco sujeitas a erros. Assim, não se implementou nenhuma tolerância a falhas e, caso a rede ou uma máquina falhe durante a execução, o processamento não poderá continuar e o trabalho feito até então será perdido. Realmente, uma tarefa como a indexação, que executa durante um período de tempo determinado e depois termina, não necessita da mesma preocupação com tolerância a falhas que um sistema de processamento de consultas, que deve ficar disponível durante o maior tempo possível. Durante a realização dos experimentos deste trabalho, não houve nenhuma falha que comprometesse uma execução dos algoritmos.

Por fim, não se considera a possibilidade das máquinas serem multiprocessadas, isto é, os algoritmos não tiram proveito especial de máquinas contendo mais de um processador, como as máquinas MIMD *Pentium Intel* e *Sun Sparc*, hoje comuns. Entretanto, os mesmos algoritmos distribuídos propostos nas seções a seguir podem ser facilmente estendidos para utilizar ao máximo o paralelismo desse tipo de máquina, bastando-se criar mais *threads* de execução, uma para cada processador. O Capítulo 6 descreve essa e outras melhorias e extensões para o presente trabalho.

3.1.2 Organização do arquivo invertido distribuído

São conhecidas duas organizações para arquivos invertidos distribuídos, introduzidas em [80]: índices locais, na qual cada máquina possui um arquivo invertido independente para sua coleção local, e índices globais, com cada máquina contendo listas invertidas completas (globais) para um subconjunto dos termos da coleção. A seguir, discutem-se em maior detalhe essas duas alternativas e seu impacto sobre a indexação e o processamento de consultas. Para a análise, considera-se que as consultas são processadas por uma máquina *broker* central, que insere as requisições em uma fila e as processa a partir daí. Esse *broker* tem acesso às listas invertidas presentes nas demais máquinas da rede através de processos servidores nessas máquinas.

A organização por arquivos invertidos locais é trivial e intuitiva. Cada máquina constrói o arquivo invertido para seu texto local, executando algum algoritmo seqüencial eficiente, como aquele descrito no Capítulo 2. Não há necessidade, portanto, de interação entre as máquinas durante o processo de indexação e uma máquina não possui informação alguma sobre os índices presentes nas demais, conhecendo apenas seu próprio arquivo invertido local.

O processamento de uma consulta nessa arquitetura acontece da seguinte forma. O *broker* recebe a consulta e a envia para todas as máquinas da rede. Cada máquina processa a consulta, obtendo uma lista de documentos-resposta, relativa a seu índice local. As máquinas então enviam suas respostas para o *broker*, que deve então produzir a resposta final e retorná-la para o usuário. Para consultas lógicas, determinar a resposta final é bem simples, bastando ao *broker* realizar a união das listas de documentos recebidas.

Entretanto, para consultas ordenadas através do modelo vetorial, a organização por índices locais torna-se desfavorável. Após o processamento local, cada máquina deve enviar apenas

os documentos no topo de seu *ranking* para o *broker*, caso contrário a rede seria inundada de dados que acabariam por não influenciar na resposta final. O *broker* recebe as respostas das máquinas e, por um processo de intercalação, produz o conjunto-resposta final. Entretanto, como nenhuma máquina (nem mesmo o *broker*) possui informação global sobre a coleção, a ordenação segundo os critérios do modelo vetorial fica prejudicada pelo desconhecimento de valores como a freqüência de um termo na coleção como um todo, ou o tamanho de um documento em relação aos demais.

O processamento distribuído de consultas pelo modelo vetorial requer, portanto, informação global sobre a coleção, de forma a não degradar a qualidade das respostas. Uma forma de se fazer isso é construir índices invertidos globais. As listas invertidas reportariam então *ocorrências dos termos em todos os documentos da coleção, não importa em que máquina esses documentos estivessem armazenados. O arquivo invertido global seria distribuído* entre as máquinas baseando-se nos termos e não mais nos documentos, ou seja, cada máquina armazenaria as listas invertidas para um subconjunto dos termos da coleção e não para um subconjunto dos documentos.

Assim, ao processar uma consulta sobre um arquivo invertido global, o *broker*, ciente de qual máquina contém a lista para cada termo, aciona somente aquelas máquinas que abrigam as listas dos termos presentes na consulta. Logo, para uma dada consulta, pode ocorrer que nem todas as máquinas participem da execução. Cada máquina ativa processa uma parte distinta da consulta, e retorna para o *broker* uma lista parcial de documentos-resposta. Este último, de posse da informação global sobre a coleção, pode intercalar as respostas parciais e produzir uma ordenação final de melhor qualidade que aquela proporcionada pela organização local.

Além de possibilitar a obtenção de uma resposta de melhor qualidade, a organização global proporciona também uma maior eficiência no processamento distribuído de consultas, como mostra a Tabela 3.1, apresentada em [63]. Esses resultados correspondem a valores estimados por um modelo analítico baseado em experimentos realizados sobre índices locais e globais em uma rede de 80 Mbps. Portanto, para redes rápidas, o desempenho dos índices globais é superior, vantagem que se acentua à medida em que aumentam fatores como a própria velocidade da rede, a taxa de transferência dos discos, o número de processadores e a sua velocidade. Como essas são tendências claras de evolução dos equipamentos, acredita-se que a organização global é a alternativa a ser escolhida.

Uma justificativa qualitativa para a superioridade da organização global dos índices é apresentada em [63] e incluída a seguir. Na organização local, cada máquina deve processar todos os termos da consulta. Isso faz com que cada uma tenha que ler as listas invertidas (ou ao menos parte delas) de todos termos da consulta. Como essas listas encontram-se armazenadas em partes distintas do índice, o número de *seeks* a disco realizados é considerável. Sendo essa uma operação cara, o tempo de resposta de cada máquina é alto. Já na organização global, a consulta é repartida entre as máquinas, com isso, cada uma deve processar as listas de uma quantidade

Número de processadores	Tempo de resposta (s)	
	Local	Global
2	21.26	19.64
4	17.13	11.13
8	14.58	8.86
16	13.11	7.50
32	12.23	7.00
64	11.78	9.93

Tabela 3.1: Tempo total estimado para 50 consultas TREC sobre índices locais e globais.

menor de termos. Para consultas pequenas (como ocorre na *Web*), o caso mais freqüente é cada máquina processar a lista de apenas um termo. Nesse caso, os dados lidos estão em posições contíguas do disco, o número de *seeks* realizados é pequeno e o tempo de resposta de cada máquina é menor.

Devido a esses fatores, a organização considerada neste trabalho é baseada em índices globais. Ao contrário da geração de arquivos invertidos locais, para a qual basta a simples execução de algum algoritmo seqüencial eficiente, o processo de construção de arquivos globais requer maior engenharia e a abordagem mais intuitiva pode ser significativamente melhorada, como mostram os resultados deste trabalho. Nas seções seguintes, serão apresentados os algoritmos distribuídos propostos para a geração de arquivos invertidos globais.

3.2 Computação do Vocabulário Global

A família de algoritmos distribuídos desenvolvida neste trabalho baseia-se no conhecimento prévio do vocabulário global por todas as máquinas envolvidas no processo de indexação. O **vocabulário global** de uma coleção distribuída corresponde ao conjunto dos termos distintos presentes no texto como um todo. Assim, todos os algoritmos distribuídos possuem em comum a primeira fase, o levantamento do vocabulário global, apresentado nesta seção.

Visão geral do processo. As máquinas participantes da indexação realizam, em paralelo, a leitura e *parsing* de suas coleções, extraindo seus respectivos vocabulários locais, da mesma forma que a fase A do algoritmo seqüencial descrito na Seção 2.1.4.

As máquinas, numeradas de 0 a $p-1$, são então emparelhadas duas a duas e os processadores de índice ímpar enviam seu vocabulário local para aqueles de índice par imediatamente inferior ao seu, como mostra a parte inferior da Figura 3.2. Assim, o processador 1 envia seu vocabulário para o processador 0, o de índice 3 envia para o de índice 2 e assim por diante. Os processadores de índice par então intercalam o vocabulário recebido com seu vocabulário

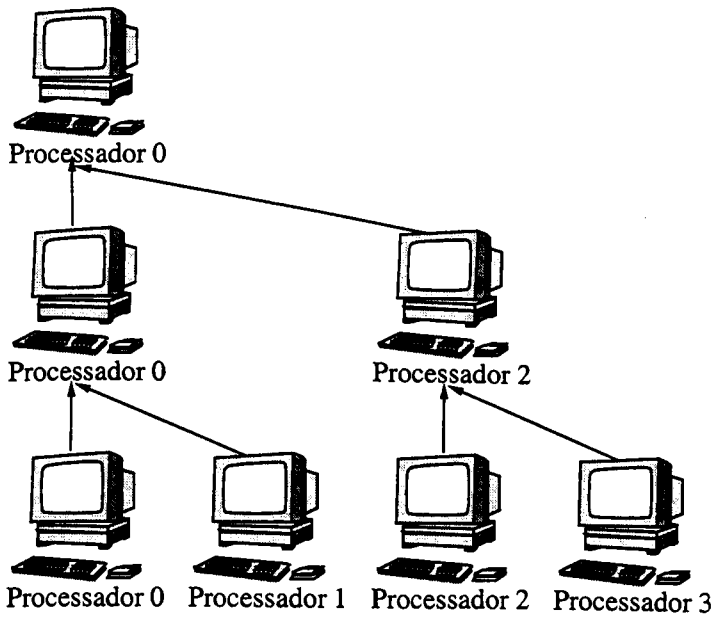


Figura 3.2: Computação do vocabulário global com 4 processadores.

local e o processo de emparelhamento reinicia, saltando-se agora as máquinas de índice ímpar. Dessa forma, o processador 2 envia o vocabulário nele contido para a máquina 0, como ilustrado na parte superior da Figura 3.2. Caso houvesse mais máquinas no exemplo, a de índice 4 receberia os dados da de índice 6 e assim por diante. A máquina que recebe um vocabulário sempre o intercala com os termos locais. Esse processo se repete até que todo o vocabulário esteja armazenado na máquina de índice 0.

Esse algoritmo é semelhante ao utilizado em [68] para a obtenção do vocabulário global e baseia-se nos algoritmos PRAM de redução em paralelo apresentados em [59]. O processo toma $\lceil \log p \rceil$ passos, se p é o número de máquinas envolvidas na execução.

O processador 0 então gera uma função *hash* perfeita para o vocabulário global e a distribui, juntamente com as *string* desse vocabulário, às demais máquinas. Os algoritmos distribuídos utilizarão a função *hash* perfeita para identificar os termos da coleção através de inteiros únicos, globais a todas elas.

Representação algorítmica. Uma maior formalização do processo de levantamento do vocabulário global é apresentada pelo Algoritmo 3.1. A etapa de transmissão e intercalação dos vocabulários descrita acima corresponde à Subfase 3, na qual se utiliza o mesmo algebrismo de [59] para determinar, a cada passo do processo, quais máquinas devem enviar seu vocabulário e quais devem receber.

Modelo matemático. A Equação 3.1 traz a análise matemática dos custos da fase de computação do vocabulário global. Nessa equação, o tempo referido como t_{GV_1} corresponde à leitura

Subfase 1: **para cada** processador π, $0 \leq \pi \leq p-1$, em paralelo **faça**
  para cada $d_j \in C_\pi$ **faça**
    1.1) *Leia*(d_j)
    1.2) **para cada** $t_i \in d_j$ **faça**
      $V_\pi \leftarrow V_\pi \cup \{t_i\}$
    fim para
  fim para
fim para

Subfase 2: *Ordene*(V)

Subfase 3: **para cada** processador π, $0 \leq \pi \leq p-1$, em paralelo **faça**
  para $j \leftarrow 0$ **até** $\lceil \log p \rceil - 1$ **faça**
    se $\pi \bmod 2^{j+1} = 0 \wedge \pi + 2^j < p$ **então**
      $V_\pi \leftarrow V_\pi \cup V_{\pi+2^j}$
    fim se
  fim para
fim para

Subfase 4: No processador 0: $h_{perf} \leftarrow$ *ConstruaHashingPerfeito*(V_0)

Subfase 5: Processador 0 envia V_0 e h_{perf} para os demais

Algoritmo 3.1: Computação do vocabulário global de uma coleção distribuída.

e ao *parsing* do texto, enquanto que t_{GV_2} representa a ordenação do vocabulário, assim como as fases A1 e A2 da Equação 2.1.

$$
\begin{aligned}
t_{GV} &= t_{GV_1} + t_{GV_2} + t_{GV_3} + t_{GV_4} + t_{GV_5} \\
\\
t_{GV_1} &= t_{GV_{1.1}} + t_{GV_{1.2}} = n(t_r + t_p) \\
t_{GV_2} &= (1.2v \log v)|w_V| t_{cs} \\
t_{GV_3} &= \left[\sum_{i=0}^{\lceil \log p \rceil - 1} K(2^i n)^\beta\right] \times |w_V|(t_n + t_{cs}) \\
&= K[(p-1)n]^\beta \times |w_V|(t_n + t_{cs}) \\
&= v(p-1)^\beta |w_V|(t_n + t_{cs}) \\
t_{GV_4} &= O(v(p-1)^\beta) \\
t_{GV_5} &= v(p-1)^\beta(|w_V| + 4\alpha^{-1})t_n
\end{aligned}
\tag{3.1}
$$

A dedução do custo t_{GV_3} requer algumas manipulações algébricas. Na primeira etapa da intercalação dos vocabulários, o número de termos transferidos entre duas máquinas da rede é dado por Kn^β, que equivale a v, o tamanho do vocabulário local a uma máquina. Logo, o volume de dados transmitido é $Kn^\beta \times |w_V|$, se $|w_V|$ é o tamanho médio das palavras do vocabulário. Nesse ponto supõe-se a alta capacidade da rede, pois não se considera a possibilidade

de contenção na comunicação entre dois diferentes pares de máquinas.

Na segunda etapa da intercalação, o número de termos transferidos entre pares de máquinas é $K(2n)^{\beta}$, correspondente ao vocabulário formado por um texto de $2n$ bytes. Esse valor é inferior a $2 \times Kn^{\beta}$, pois β é menor que 1, isto é, o vocabulário cresce sublinearmente com o tamanho do texto. Como a etapa de intercalação toma $\lceil \log p \rceil$ passos, o número total de termos transmitidos é dado pelo somatório mostrado, que pode ser simplificado para $K[(p-1)n]^{\beta}$ ou, lembrando-se que $Kn^{\beta} = v$, o vocabulário local a um processador, $v(p-1)^{\beta}$.

O custo da subfase 4 corresponde à construção, na máquina 0, de uma função *hash* perfeita para o vocabulário global, enquanto que o valor t_{GV_5} contabiliza o custo de transmiti-la, juntamente com as *strings* desse vocabulário, às demais máquinas da rede. Para cada termo do vocabulário, a função *hash* perfeita corresponde a uma sobrecarga de $4\alpha^{-1}$ bytes (o mapeamento g), onde α^{-1} é o inverso do fator de preenchimento discutido na Seção 2.2.2. O valor 4 vem do fato de se usarem inteiros de 4 bytes na tabela que compõe a função.

Com cada máquina de posse do vocabulário global da coleção, pode-se determinar qual máquina será responsável pelo armazenamento da lista de cada termo. Isso é feito de uma forma bem simples mas que, segundo [7, 63], proporciona boa divisão do arquivo invertido em termos de tamanho e carga de consultas. Para um vocabulário global de tamanho v_g e p máquinas, a máquina 0 fica com as listas dos termos 0 a $v_g/p - 1$, a máquina 1 com os termos v_g/p a $2v_g/p - 1$ e assim por diante, com a máquina π armazenando as listas $\pi v_g/p$ a $(\pi+1)v_g/p - 1$, até a máquina $p-1$, que será responsável pelos termos $(p-1)v_g/p$ a $v_g - 1$.

3.3 O Algoritmo LL: *Buffers* Locais e Listas Locais

Passa-se agora à descrição do primeira alternativa para indexação distribuída proposta neste trabalho. Esse algoritmo, denominado LL (de *buffers* locais e listas locais, como justificado a seguir), consiste na abordagem mais intuitiva para a geração de arquivos invertidos globais. Ele propõe que, após a fase de determinação do vocabulário global, sejam geradas as listas invertidas locais a cada processador, utilizando o algoritmo seqüencial proposto no Capítulo 2, e somente então troquem-se as listas entre as máquinas para a geração do índice global. Aí está a razão para o nome LL: as triplas identificadas no texto local, mesmo que pertençam a listas que serão armazenadas em outra máquina, são inseridas no *buffer* local da máquina que as identificou e depois gravadas em uma lista também local a essa máquina.

Logo, de acordo com o algoritmo LL, somente depois de construídos os índices locais, cada máquina toma as listas que não se destinam a ela e as envia para a máquina devida. Por outro lado, cada máquina também recebe trechos das listas destinadas a ela e os intercala com os trechos obtidos de seu texto local, produzindo assim sua parte do arquivo invertido global.

A Figura 3.3 ilustra o funcionamento geral do algoritmo LL em relação ao esquema de execução do algoritmo seqüencial apresentado na Figura 2.5. O princípio básico é construir os

arquivos invertidos locais através do algoritmo seqüencial e então trocar os trechos das listas entre as máquinas. Dessa forma, a troca de dados entre as máquinas participantes ocorre após a produção dos arquivos invertidos locais, o que é ilustrado pela Figura 3.3 mostrando-se a troca de informações com a rede logo antes da obtenção do arquivo invertido final. Na realidade, outro índice temporário é produzido entre os dois arquivos representados nessa figura.

Visão geral do processo. Assim como o algoritmo seqüencial descrito na Seção 2.1.4, os algoritmos distribuídos neste capítulo podem ser decompostos em 3 fases principais: o levantamento do vocabulário, a construção dos índices e a intercalação dos *runs* produzidos, gerando o arquivo invertido final. Baseado neste esquema, o algoritmo LL é descrito a seguir.

Como já citado, a primeira fase do algoritmo LL corresponde ao levantamento do vocabulário global, descrito na Seção 3.2. A segunda fase, por sua vez, é idêntica à indexação seqüencial apresentada na Seção 2.1.4, com uma pequena modificação. A intercalação dos *runs* produzidos não gera duplas $\langle f, d \rangle$, mas mantém os componentes t das triplas, pois estes serão necessários para a intercalação final, que gera as listas globais.

Com isso, a intercalação pode ser reescrita como $F'' \leftarrow \mathit{Intercale}(F', R)$, onde F'' é um segundo arquivo temporário, contendo o índice local para cada máquina e mantendo os componentes t das triplas. O tamanho f'' desse índice é intermediário aos valores f e f'' (os tamanhos dos arquivos final e temporário da indexação seqüencial): ele é maior que f, pois deve ainda armazenar os índices dos termos, porém é menor que f', pois os *runs* já foram intercalados e, logo, os intervalos entre termos, freqüências e documentos são menores, o que possibilita melhor compressão. A segunda fase do algoritmo LL encontra-se ilustrada na Figura 3.4(a).

Ocorre então a troca e a nova intercalação das listas invertidas locais produzidas na fase anterior, quando os trechos das listas locais a cada processador são enviados à máquina de destino. Como descrito na Seção 4.5, cada máquina executa uma *thread* para envio e outra para a recepção das listas, permitindo que ambos os processos sejam realizados concorrentemente, sem riscos de *deadlocks*. Terminada a etapa de troca de listas, o arquivo temporário f'' contém p *runs* ordenados. Esses *runs* são intercalados através de um *heap* de tamanho p, produzindo o arquivo invertido final F. Essa fase é ilustrada pela Figura 3.4(b).

Representação algorítmica. O Algoritmo 3.2 esquematiza o algoritmo distribuído LL. A fase A é dada pelo Algoritmo 3.1, enquanto que a fase B corresponde às fases B e C do Algoritmo 2.1. O arquivo invertido local F'' formado em uma máquina π pode ser visto como uma seqüência de p conjuntos de listas, cada um destinado a uma diferente máquina, sendo representado por

$$F''_{\pi} \equiv \langle L_{\pi \to 0}, L_{\pi \to 1}, \ldots, L_{\pi \to \mu}, \ldots, L_{\pi \to p} \rangle.$$

Na fase C, cada processador π lê de seu arquivo temporário F''_{π} e envia pela rede as listas

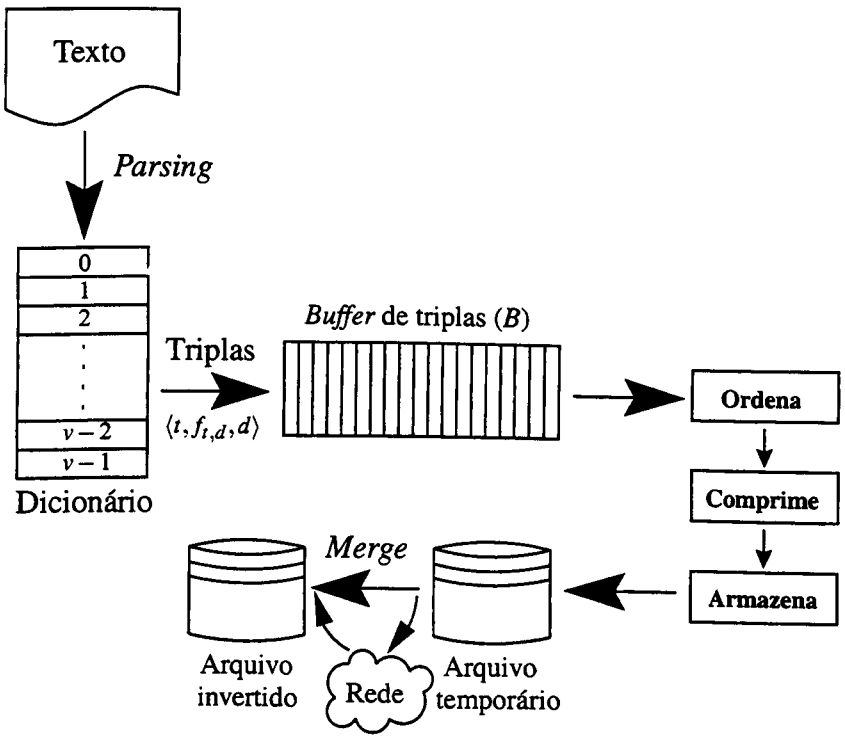
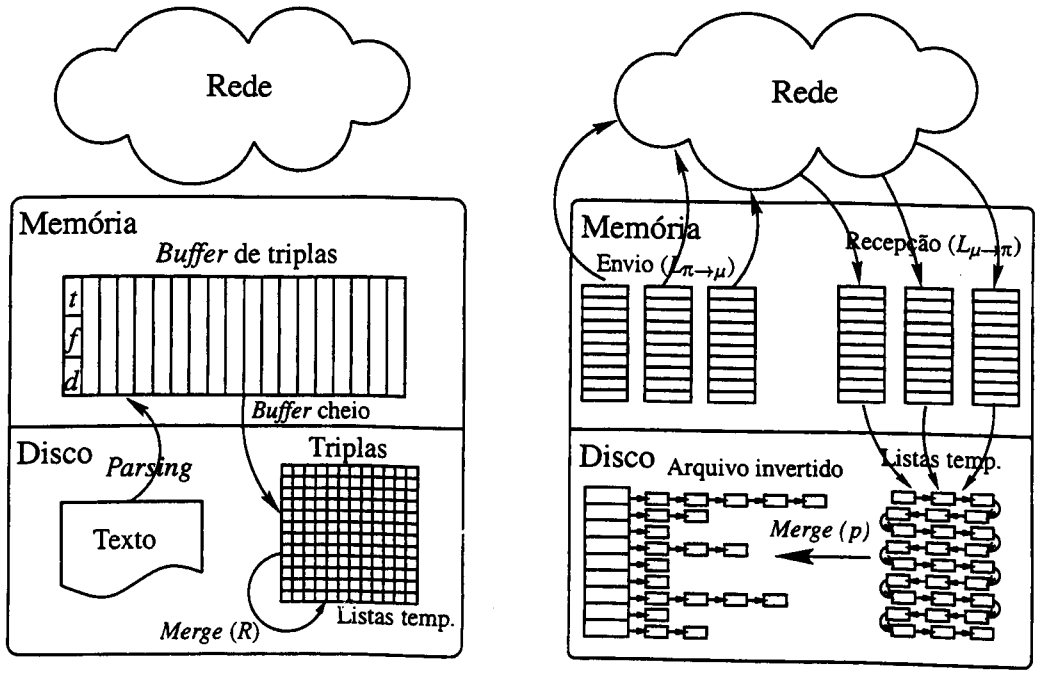


Figura 3.3: Esquema simplificado de execução do algoritmo LL.



(a) Geração das listas locais

(b) Troca das listas e intercalação

Figura 3.4: Funcionamento detalhado do algoritmo LL.

Fase A: *Obtenção do vocabulário global.* Executada pelo Algoritmo 3.1.

Fase B: *Geração das listas invertidas locais.* Executada pelas fases B e C do Algoritmo 2.1, com a diferença que os arquivos finais (locais) F'' mantém os componentes t.

Fase C: *Troca e intercalação das listas.*

para cada processador π, $0 \leq \pi \leq p-1$, em paralelo **faça**
  para $\mu \leftarrow 0$ **até** $p-1$ e $\mu \neq \pi$ **faça**
    C1) $Leia(L_{\pi\rightarrow\mu}, F''_{\pi})$
    C2) $Troque(L_{\pi\rightarrow\mu}, L_{\mu\rightarrow\pi})$
    C3) $Grave(L_{\mu\rightarrow\pi}, F''_{\pi})$
  fim para
  C4) $F_{\pi} \leftarrow Intercale(F''_{\pi}, p)$
fim para

Algoritmo 3.2: Algoritmo distribuído LL.

$L_{\pi\rightarrow\mu}$ destinadas a cada outro processador μ. Tais listas são chamadas "emigrantes" em relação a π, pois foram geradas pelo texto local, mas serão armazenadas em outra máquina. Por outro lado, π também recebe de μ as listas "imigrantes" $L_{\mu\rightarrow\pi}$.

Modelo matemático. O custo do Algoritmo 3.2 é expresso pela Equação 3.2. A fase A tem o custo dado pela Equação 3.1 e a fase B corresponde aos termos t_{SEQ_B} e t_{SEQ_C} da Equação 2.1, com a ressalva, destacada por $\star$, de que o tamanho do arquivo produzido pela intercalação é f'' e não f. O custo da fase C pode ser totalizado a pela leitura, transmissão e escrita das listas que devem ser trocadas entre as máquinas, correspondentes à proporção $\frac{p-1}{p}$ do índice temporário. Por fim, $t_{LL_{C4}}$ contabiliza a intercalação dos p *runs*.

$$
\begin{aligned}
t_{LL} &= t_{LL_A} + t_{LL_B} + t_{LL_C} \\
\\
t_{LL_A} &= t_{GV} \\
t_{LL_B} &= t_{SEQ_B} + t^{\star}_{SEQ_C} \\
\\
t_{LL_C} &= t_{LL_{C1}} + t_{LL_{C2}} + t_{LL_{C3}} + t_{LL_{C4}} \\
t_{LL_{C1}} &= \tfrac{p-1}{p} f'' t_r \\
t_{LL_{C2}} &= \tfrac{p-1}{p} f'' t_n \\
t_{LL_{C3}} &= \tfrac{p-1}{p} f'' t_r \\
t_{LL_{C4}} &= f'' \left(\tfrac{p-1}{pM} t_s + t_r + t'_z \right) + x\lceil \log p \rceil t_{ct} + f(t_r + t_z) \\
t_{LL_C} &= \tfrac{p-1}{p} f''(2t_r + t_n) + f'' \left(\tfrac{p-1}{pM} t_s + t_r + t'_z \right) + x\lceil \log p \rceil t_{ct} + f(t_r + t_z)
\end{aligned}
\tag{3.2}
$$

3.4 O Algoritmo LR: *Buffers* Locais e Listas Remotas

A abordagem LL, embora intuitiva, realiza muito trabalho desnecessário armazenando em listas locais aquelas triplas que se destinam a outras máquinas. Dado que um conjunto de triplas não se destina à máquina local, ele pode ser enviado pela rede tão logo seja formado. Isso acontece sempre que o *buffer* de memória torna-se cheio e é ordenado (e também comprimido), evitando, assim, uma escrita e posterior leitura de todo o arquivo invertido em disco, como o faz o algoritmo LL.

Essa é a proposta do algoritmo LR (de *buffers* locais, listas remotas). Agora, as triplas que não serão armazenadas na máquina local ainda são inseridas no seu próprio *buffer* de memória, mas não são copiadas para seu disco, de forma que a formação das listas invertidas ocorre remotamente. A Figura 3.5 ilustra a idéia geral do algoritmo LR: o envio dos dados pela rede ocorre em um passo anterior em relação ao LL, assim que as triplas são ordenadas e comprimidas.

Visão geral do processo. As três fases do algoritmo LR diferem bastante daquelas da abordagem LL pelo fato do envio de dados pela rede ser interposto com o processamento do texto local. Nos parágrafos seguintes, descreve-se o funcionamento do algoritmo LR.

Como acontece com os demais algoritmos, a primeira fase corresponde ao levantamento do vocabulário global, descrito na Seção 3.2. Passa-se então ao processamento do texto local, intercalado com a troca de triplas, como mostrado na Figura 3.6(a). O funcionamento dessa fase pode ser também comparado com a abordagem seqüencial do Algoritmo 2.1. Realiza-se o *parsing* do texto, armazenando-se todas as triplas identificadas no *buffer* de memória local B. Quando este se torna cheio, ele é ordenado e comprimido formando-se um *run*, assim como no algoritmo seqüencial.

Contudo, em vez de ser totalmente armazenado em disco, o *run* é visto como uma seqüência de p "sub-*runs*", cada um destinado a uma das máquinas executando o algoritmo. Os sub-*runs* produzidos na máquina local são então enviados a seus destinos, da mesma forma que, durante todo o processo, sub-*runs* produzidos em outras máquinas são recebidos pela rede. A concorrência entre o tratamento dos dados recebidos pela rede e o processamento do texto é obtido através de *multithreading*, como descrito na Seção 4.5.2. Sempre que um sub-*run* chega a uma máquina, ele é diretamente armazenado em disco, devendo ser intercalado aos demais na fase seguinte.

Enfim, a fase C procede a intercalação dos sub-*runs* produzidos anteriormente e encontra-se ilustrada na Figura 3.6(b). Como supõe-se que as máquinas possuem a mesma quantidade de memória M e que o texto encontra-se igualmente distribuído, cada uma produz R *runs* no processamento de seu texto local. Como cada *run* é particionado em p partes, as máquinas terminam a fase B contendo $p \times R$ sub-*runs* cada. Portanto, a intercalação que ocorre na fase C é de $p \times R$ caminhos e, logo, mais cara, pois requer mais *seeks* a disco.

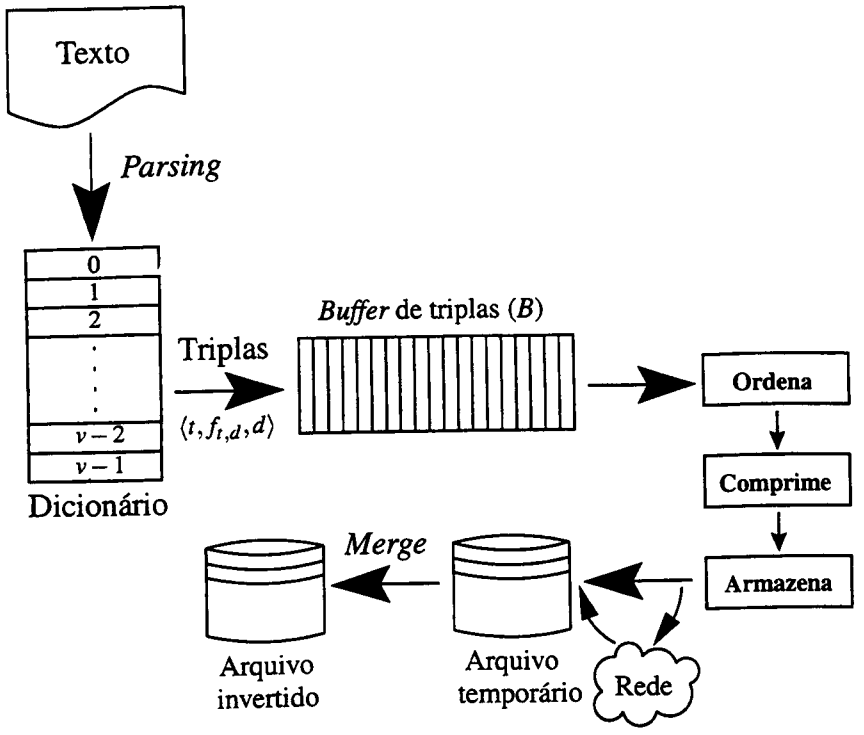
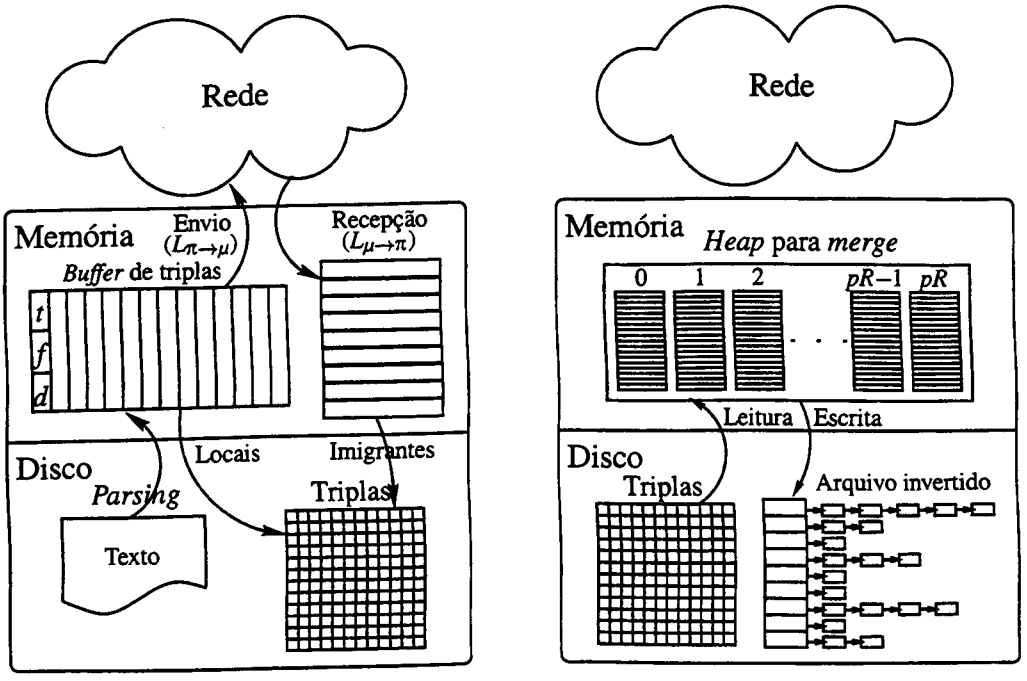


Figura 3.5: Esquema simplificado de execução do algoritmo LR.



(a) Processamento do texto e troca de triplas (sempre que o *buffer* se torna cheio).

(b) Intercalação de $p \times R$ *runs*.

Figura 3.6: Funcionamento detalhado do algoritmo LR.

Representação algorítmica. O processo anteriormente descrito é melhor formalizado pelo Algoritmo 3.3. A fase B1 é idêntica à correspondente do Algoritmo 2.1. A cada vez que o *buffer B* excede a capacidade M da memória, ele é ordenado e comprimido, passando a ser visto como um conjunto de sub-listas L

$$B_\pi \equiv \langle L_{\pi\to 0}, L_{\pi\to 1}, \ldots, L_{\pi\to\mu}, \ldots, L_{\pi\to p}\rangle.$$

As sub-listas emigrantes $L_{\pi\to\mu}$ são enviadas pela rede, enquanto que, concorrentemente, as sub-listas imigrantes são recebidas e gravadas em disco. As sub-listas produzidas localmente também são diretamente armazenadas em disco. Como não há contenção no acesso à rede, esse processo foi representado pela função *Troque*(), de forma que ele seja contabilizado apenas uma vez para cada par de máquinas.

Modelo matemático. A Equação 3.3 expressa os custos do algoritmo LR. Destaca-se o termo $t_{\mathsf{LR}_{B2.3}}$, que contabiliza o tempo de envio das sub-listas pela rede. Como elas são comprimidas, o valor é representado em relação ao tamanho f' do arquivo temporário. Apenas uma proporção equivalente a $(p-1)/p$ é enviada pela rede, pois $1/p$ dos dados ficam na máquina local. Por outro lado, o volume de dados gravado em disco durante a fase B corresponde a todo o arquivo temporário, como destacado pelo termo $t_{\mathsf{LR}_{B2.4}} = f' t_r$. Isso ocorre porque tanto os dados que ficam na máquina local ($f'\frac{1}{p}$) como aqueles recebidos de outras máquinas ($f'\frac{p-1}{p}$) devem ser gravados no arquivo temporário durante a fase B.

Cumpre também ressaltar os termos $f'\frac{pR-1}{pRM}t_s$ e $x\lceil \log pR\rceil t_{ct}$, relativos, respectivamente, ao número de *seeks* e comparações de triplas na fase de intercalação. Como eles são dependentes do produto pR, a fase C do algoritmo LR é mais cara que a dos demais algoritmos.

$$\begin{aligned}
t_{\mathsf{LR}} &= t_{\mathsf{LR}_A} + t_{\mathsf{LR}_B} + t_{\mathsf{LR}_C} \\
t_{\mathsf{LR}_A} &= t_{GV} \\
t_{\mathsf{LR}_B} &= t_{\mathsf{LR}_{B1}} + t_{\mathsf{LR}_{B2}} \\
t_{\mathsf{LR}_{B1}} &= t_{\mathsf{LR}_{B1.1}} + t_{\mathsf{LR}_{B1.2}} = n(t_r + t'_p) \\
t_{\mathsf{LR}_{B2}} &= t_{\mathsf{LR}_{B2.1}} + t_{\mathsf{LR}_{B2.2}} + t_{\mathsf{LR}_{B2.3}} + t_{\mathsf{LR}_{B2.4}} \\
t_{\mathsf{LR}_{B2}} &= \lambda x t_{ct} + f' t_z + f'\tfrac{p-1}{p} t_n + f' t_r \\
t_{\mathsf{LR}_B} &= n(t_r + t'_p) + \lambda x t_{ct} + f'(t_z + \tfrac{p-1}{p} t_n + t_r) \\
t_{\mathsf{LR}_C} &= f'\left(\tfrac{pR-1}{pRM} t_s + t_r + t'_z\right) + x\lceil \log pR\rceil t_{ct} + f(t_r + t_z)
\end{aligned} \tag{3.3}$$

Fase A: *Obtenção do vocabulário global.* Executada pelo Algoritmo 3.1.

Fase B: *Processamento do texto e troca de triplas.*

para cada processador π, $0 \leq \pi \leq p-1$, em paralelo **faça**
  B1) **para cada** $d_j \in C_\pi$ **faça**
    B1.1) *Leia*(d_j)
    B1.2) **para cada** $t_i \in d_j$ **faça**
      $B_\pi \leftarrow B_\pi \cup \langle t_i, d_j, f_{i,j} \rangle$
    fim para
  fim para
  B2) **se** $|B_\pi| \times |\langle t, f, d \rangle| \geq M$ **então**
    B2.1) *Ordene*(B_π) /* $t\uparrow, f\downarrow, d\uparrow$ */
    B2.2) *Comprima*(B_π)
    para $\mu \leftarrow 0$ **até** $p-1$ **faça**
      se $\mu \neq \pi$ **então**
        B2.3) *Troque*$(L_{\pi\to\mu}, L_{\mu\to\pi})$
      fim se
      B2.4) *Grave*$(L_{\mu\to\pi}, F'_\pi)$
        $R_\pi \leftarrow R_\pi + 1$
    fim para
    $B \leftarrow \emptyset$
  fim se
fim para

Fase C: *Intercalação das listas.*

para cada processador π, $0 \leq \pi \leq p-1$, em paralelo **faça**
  $F_\pi \leftarrow$ *Intercale*(F'_π, R_π) /* $R_\pi = p \times R$ */
fim para

Algoritmo 3.3: Algoritmo distribuído LR.

3.5 O Algoritmo RR: *Buffers* Remotos e Listas Remotas

Uma terceira abordagem pode ser proposta, bastando observar que as triplas emigrantes podem ser enviadas pela rede ainda mais cedo, tão logo elas sejam identificadas. Dessa forma, elas não seriam armazenadas nem mesmo no *buffer* local da máquina onde foram produzidas, indo diretamente para a memória da máquina remota. A partir daí, o funcionamento do algoritmo passaria a ser idêntico ao da abordagem seqüencial, com cada máquina ordenando, comprimindo e gravando o conteúdo de seus *buffers* em seus próprios discos. A intercalação dos *runs* após o processamento do texto também se daria da mesma forma que no algoritmo seqüencial.

Devido ao fato das triplas emigrantes serem remetidas tão logo sejam descobertas, sendo armazenadas no *buffer* da máquina remota, onde também são geradas as listas invertidas finais, essa abordagem foi denominada algoritmo RR, de *buffers* remotos e listas remotas. A Figura 3.7 ilustra o princípio básico desse algoritmo: assim que as triplas são identificadas (sendo removidas da estrutura de dicionário) elas são diretamente enviadas pela rede. Da mesma forma, triplas imigrantes são recebidas ao longo de todo o processamento do texto e armazenadas no próprio *buffer* de memória da máquina, juntamente com as demais triplas.

Visão geral do processo. O algoritmo RR tem sua divisão em fases bastante semelhante à do algoritmo LR: uma primeira fase de obtenção do vocabulário global, seguida do processamento do texto com a troca de triplas e da intercalação produzindo o arquivo invertido final. As diferenças em relação à abordagem LR e maiores detalhes são descritos a seguir, enquanto que o funcionamento é ilustrado na Figura 3.8.

Após o levantamento do vocabulário global, passa-se ao processamento do texto para a obtenção das triplas relativas ao texto da máquina local π. Assim que uma tripla $\langle t_i, f_{i,j}, d_j \rangle$ é identificada, verifica-se em qual máquina a lista de t_i será armazenada. Se essa máquina é μ, tal que $\mu \neq \pi$, a tripla é emigrante e deve ser enviada para μ. Para que a rede não seja saturada com um número muito grande de pequenos pacotes de dados (um para cada tripla), as triplas são copiadas para blocos de dados, descritos na Seção 4.4.3, e somente então enviados pela rede. Por outro lado, se $\mu = \pi$, a tripla é local e deve ser armazenada no *buffer* local B_π.

Concorrentemente, triplas imigrantes são recebidas das demais máquinas e também inseridas em B_π. A implementação dessa concorrência também utiliza *multithreading* como os outros algoritmos distribuídos, todavia, ela requer maiores cuidados. Há uma *thread* para receber os dados de cada outra máquina μ, e todas procuram inserir as triplas que recebem em B_π, assim como também o faz a *thread* que lê e processa o texto local. Há, portanto, p *threads* buscando acesso simultâneo a uma mesma área de memória compartilhada. Para garantir a consistência da estrutura de B_π, utiliza-se um mecanismo de exclusão mútua, esclarecido na Seção 4.5.2, página 96.

Posto que todas as triplas inseridas no *buffer* local B_π são destinadas a permanecer na máquina π, o restante do processamento do algoritmo RR é idêntico ao algoritmo seqüencial.

Quando o *buffer* torna-se cheio, ele é ordenado, comprimido e gravado no disco local, formando um *run*. Com o fim do processamento do texto, as máquinas intercalam seus R *runs* de forma independente, produzindo os arquivos invertidos finais.

Representação algorítmica. O Algoritmo 3.4 formaliza a abordagem RR descrita anteriormente. A troca de triplas ocorre na etapa B1 e não mais na B2, como o algoritmo LR. Assim que uma tripla $\langle t_i, f_{i,j}, d_j \rangle$ é identificada, caso ela se destine a outra máquina, ela é imediatamente enviada pela rede, ao passo que triplas imigrantes são recebidas. Esse processo é representado pelo comando $Troque(\langle t,f,d\rangle_{\pi\to\mu}, \langle t,f,d\rangle_{\mu\to\pi})$, que faz uso de duas suposições: (i) qualquer par de máquinas pode se comunicar sem contenção e (ii) triplas recebidas de outras máquinas podem ser incorporadas ao fluxo de execução local sem maiores custos. A primeira suposição é válida para o ambiente de execução descrito na Seção 3.1.1, enquanto que a segunda requer uma sofisticada implementação de controle de concorrência para permitir o acesso ao *buffer* de triplas. Assim, as triplas $\langle t,f,d\rangle_{\mu\to\pi}$, sejam elas produzidas no texto local (isto é, $\mu = \pi$) ou recebidas pela rede, são igualmente armazenadas no *buffer* B_π.

Observa-se que, como as triplas são transmitidas independentemente, sem nenhuma ordenação, elas não podem ser comprimidas, como acontece nos algoritmos LL e LR. Estes últimos transmitem listas invertidas (ou trechos delas), permitindo o uso de compressão, pois os dados já foram ordenados na máquina local. Esse fato faz com que o volume de dados transferidos pela rede no algoritmo RR seja significativamente maior que nas demais abordagens.

A etapa B2 é idêntica à correspondente do algoritmo seqüencial: quando o *buffer* torna-se cheio, ele é ordenado, comprimido e gravado, produzindo-se um novo *run* e retomando-se o processamento do texto local. Ao final do processo, R *runs* serão produzidos, assim como no algoritmo seqüencial. A fase C realiza a intercalação por R caminhos em cada máquina, gerando o arquivo invertido final.

Modelo matemático. Os custos do algoritmo RR são expressos pela Equação 3.4. Destaca-se o termo $t_{RR_{B1.3}}$, que contabiliza o tempo de transmissão das triplas pela rede. Como os dados não estão ordenados e, logo, não podem ser comprimidos, o volume em bytes de dados transmitidos é dado por $\frac{p-1}{p} x |\langle t,f,d\rangle|$, isto é, a proporção das listas locais que se destinam a outras máquinas, $(p-1)/p$, multiplicada pelo número de triplas x e seu tamanho não comprimido, $|\langle t,f,d\rangle|$.

Esse termo foi desconsiderado no modelo matemático apresentado em [62], onde supôs-se que ele poderia ser ocultado por uma implementação concorrente do envio das triplas. Como logo se viu pelos experimentos realizados com o algoritmo RR, esse fator não poderia ser negligenciado. Na realidade, o argumento de que uma implementação concorrente ocultaria o tempo de transmissão dos dados pela rede poderia ser utilizado para qualquer um dos três algoritmos, pois, como se verá na Seção 4.5.2, todos foram implementados utilizando *multithreading* e técnicas de programação concorrente.

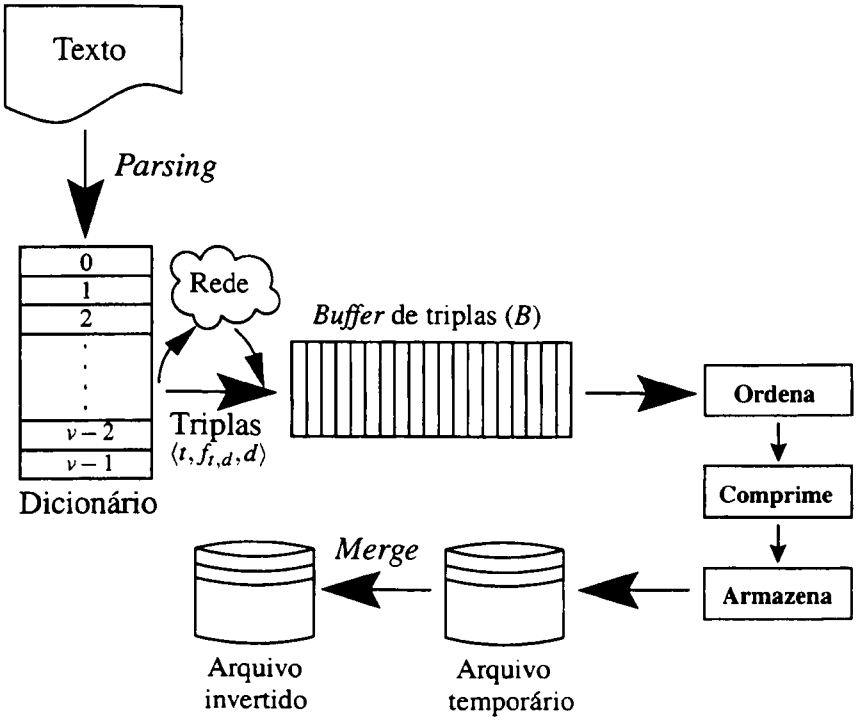
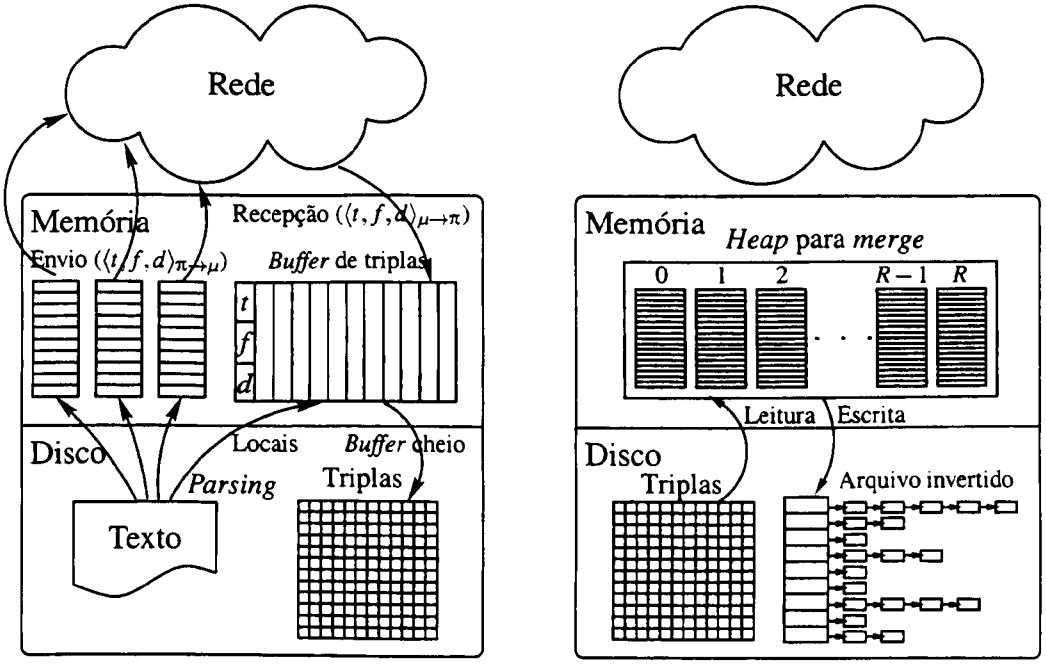


Figura 3.7: Esquema simplificado de execução do algoritmo RR.



(a) Processamento do texto e troca de triplas (ao longo de toda execução).

(b) Intercalação de *R runs*.

Figura 3.8: Funcionamento detalhado do algoritmo RR.

Fase A: *Obtenção do vocabulário global.* Executada pelo Algoritmo 3.1.

Fase B: *Processamento do texto e troca de triplas.*

para cada processador π, $0 \leq \pi \leq p-1$, em paralelo **faça**
  B1) **para cada** $d_j \in C_\pi$ **faça**
    B1.1) *Leia*(d_j)
    B1.2) **para cada** $t_i \in d_j$ **faça**
      /* Seja $t_i \in \mu$, isto é, $\langle t_i, f_{i,j}, d_j \rangle \equiv \langle t, f, d \rangle_{\pi \to \mu}$ */
      se $\mu \neq \pi$ **então**
        B1.3) *Troque*$(\langle t, f, d \rangle_{\pi \to \mu}, \langle t, f, d \rangle_{\mu \to \pi})$
      fim se
      $B_\pi \leftarrow B_\pi \cup \langle t, f, d \rangle_{\mu \to \pi}$
    fim para
  fim para
  B2) **se** $|B| \times |\langle t, f, d \rangle| \geq M$ **então**
    B2.1) *Ordene*(B) /* $t \uparrow, f \downarrow, d \uparrow$ */
    B2.2) *Comprima*(B)
    B2.3) *Grave*(B, F')
      $B \leftarrow \emptyset$
      $R_\pi \leftarrow R_\pi + 1$
  fim se
fim para

Fase C: *Intercalação das listas.*

para cada processador π, $0 \leq \pi \leq p-1$, em paralelo **faça**
  $F_\pi \leftarrow$ *Intercale*(F'_π, R_π) /* $R_\pi = R$ */
fim para

Algoritmo 3.4: Algoritmo distribuído RR.

Outra suposição que foi feita em [62] e mantida pela expressão da fase B1 na Equação 3.4 é que o custo para se incorporar triplas recebidas pela rede ao fluxo de execução local pode ser desconsiderado, isto é, as triplas imigrantes podem ser inseridas no *buffer B* com o mesmo custo que as triplas produzidas localmente. Essa suposição é expressa pelo fato do passo B1.3, no qual as triplas são trocadas pela rede, contabilizar apenas o tempo de envio e voltará a ser discutida nas Seções 4.5.2 (página 96) e 5.4.2 (página 121).

Nota-se também que o custo da fase C é idêntico ao correspondente da expressão do algoritmo seqüencial.

$$
\begin{aligned}
t_{\mathsf{RR}} &= t_{\mathsf{RR}_A} + t_{\mathsf{RR}_B} + t_{\mathsf{RR}_C} \\
t_{\mathsf{RR}_A} &= t_{GV} \\
t_{\mathsf{RR}_B} &= t_{\mathsf{RR}_{B1}} + t_{\mathsf{RR}_{B2}} \\
t_{\mathsf{RR}_{B1}} &= t_{\mathsf{RR}_{B1.1}} + t_{\mathsf{RR}_{B1.2}} + t_{\mathsf{RR}_{B1.3}} = n(t_r + t'_p) + \tfrac{p-1}{p} x |\langle t, f, d \rangle| t_n \\
t_{\mathsf{RR}_{B2}} &= t_{\mathsf{RR}_{B2.1}} + t_{\mathsf{RR}_{B2.2}} + t_{\mathsf{RR}_{B2.3}} \\
t_{\mathsf{RR}_{B2}} &= \lambda x t_{cl} + f' t_z + f' t_r \\
t_{\mathsf{RR}_B} &= n(t_r + t'_p) + \tfrac{p-1}{p} x |\langle t, f, d \rangle| t_n + \lambda x t_{cl} + f'(t_z + t_r) \\
t_{\mathsf{RR}_C} &= t_{SEQ_C}
\end{aligned}
\tag{3.4}
$$

Capítulo 4

Implementação

Neste capítulo apresentam-se maiores detalhes da implementação dos algoritmos discutidos anteriormente. O processo de geração de índices foi embutido em um pacote de software chamado **DIG**, de *Distributed Index Generation*, capaz de realizar a indexação de textos em diferentes formatos, executando de forma seqüencial ou distribuída.

Apesar da notação de classes adotada para a descrição de seus módulos, o sistema **DIG** foi implementado na linguagem C, que não oferece suporte à orientação a objetos. Essa metodologia foi na realidade utilizada no projeto do sistema e simulada na implementação através das técnicas sugeridas em [31]. Cada classe foi implementada como uma estrutura (`struct`) cujos campos não são visíveis aos clientes. Assim, um "objeto" de uma classe somente pode ser instanciado e alterado através de funções definidas no próprio código da classe. O sistema **DIG** adota a convenção de nomear essas funções no formato **Classe_método**, similarmente à sintaxe da maioria das linguagens orientadas a objeto.

C foi a linguagem de escolha por sua eficiência em tempo de execução e por sua grande integração com os sistemas operacionais da família UNIX. Esta última, por sua vez, foi escolhida para o ambiente de execução por sua robustez, estabilidade e eficiência. A variante UNIX utilizada para os experimentos foi o sistema Linux, por ser um sistema de código aberto e disponível gratuitamente para PCs.

A Seção 4.1 apresenta os diagramas de classes para as duas principais etapas do processo de indexação: a obtenção do vocabulário e a geração propriamente dita do índice. As classes que provêm suporte a ambas etapas são descritas na Seção 4.2, juntamente com outras classes de implementação não representadas nos diagramas. As classes relacionadas ao processo de levantamento do vocabulário são mostradas na Seção 4.3, enquanto que as participantes da geração do índice encontram-se detalhadas na Seção 4.4. Finalmente, a Seção 4.5 descreve como se dá o processo da geração distribuída de arquivos invertidos e os aspectos específicos da implementação de cada algoritmo distribuído.

4.1 Diagramas de Classes

Nesta seção descrevem-se os relacionamentos entre as principais classes envolvidas na implementação do sistema **DIG**. Utilizam-se para isso diagramas de colaboração, seguindo o padrão OMT (*Object Modelling Technique*), assim como feito em [65, 28]. Para cada uma das duas fases do processo, é apresentado um diagrama em separado, repetindo-se as classes em comum entre ambas. Cada diagrama possui uma classe principal, **DigHash** e **DigIndex**, que instancia as demais e controla o fluxo de execução.

A Figura 4.1 traz o diagrama de colaboração para as principais classes relacionadas à fase de levantamento do vocabulário. A seguir, apresenta-se a descrição dessas classes, de seus relacionamentos e interfaces. Maiores detalhes de implementação serão descritos nas seções seguintes.

Classe DigHash

Descrição: Controla o processo de geração da função *hash* perfeita para o texto processado.

Usa: NormHash, Parser, Option.

Função: Processar as opções para a geração especificadas pela linha de comando, inicializar as conexões da máquina local com as demais (no caso de se executar um algoritmo distribuído), disparar o processamento do texto e a obtenção do vocabulário. Armazenar as informações coletadas sobre o texto em disco, para a posterior execução da fase de indexação.

Interface:

- Construtor: Processa as opções de linha de comando, estabelece as conexões entre as máquinas participantes de um algoritmo distribuído, abre em disco os arquivos onde os resultados da execução serão armazenados e cria os principais objetos a serem utilizados no processamento.
- *parse*(): Efetua o *parsing* do texto, isto é, o processamento dos arquivos correspondendo à coleção. Os termos identificados no texto são coletados pela classe **NormHash**, descrita adiante.
- *sortVocab*(): Ordena o vocabulário obtido pelo processamento do texto, segundo a ordem lexicográfica dos termos.
- *writeVocab*(): Escreve o vocabulário já ordenado para disco ou, no caso da execução de um algoritmo distribuído, para a máquina apropriada, de acordo com os passos do Algoritmo 3.1.

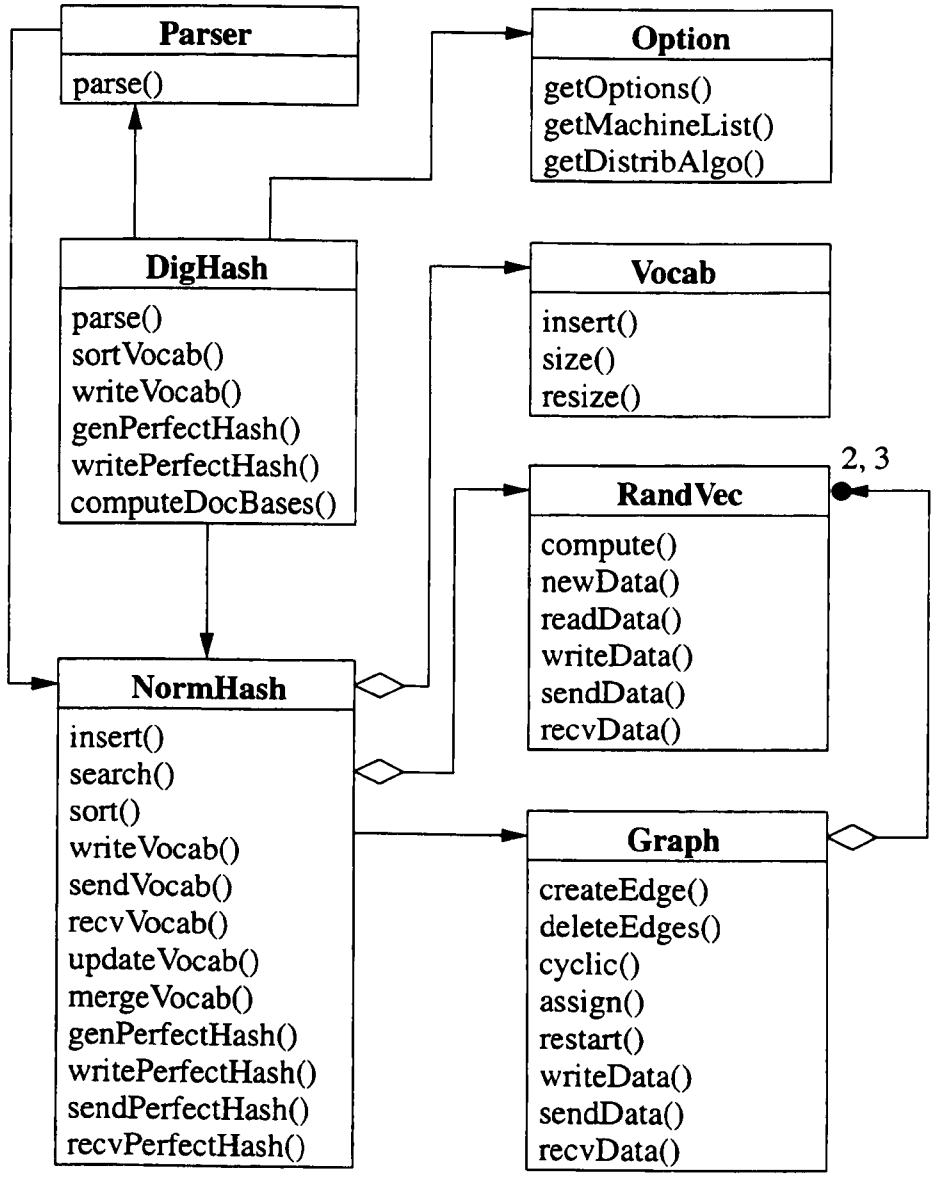


Figura 4.1: Diagrama de classes para fase de obtenção do vocabulário.

- *genPerfectHash*(): gera a função *hash* perfeita para o vocabulário obtido. Processando-se um dos algoritmos distribuídos, essa função somente é executada na máquina de índice 0.

- *writePerfectHash*(): grava em disco os dados da função *hash* perfeita gerada para a coleção. Com os algoritmos distribuídos, essa função envolve também o envio da função, pela máquina de índice 0, às demais máquinas participantes.

- *computeDocBases*(): executada pelos algoritmos distribuídos para estabelecer a partir de qual valor cada máquina irá numerar seus documentos na fase de indexação, de forma que a coleção tenha valores únicos para os identificadores de documentos. Supondo que a máquina 0 contenha c_0 documentos, a máquina 1, c_1 e assim até a máquina $p-1$, com c_{p-1} documentos, a máquina 0 numerará seus documentos a partir de 0, a máquina 1 a partir de c_0, a máquina 2 a partir de $c_0 + c_1$ e assim até a máquina $p-1$, que iniciará a numeração de seus documentos a partir do valor $\sum_{\pi=0}^{p-2} c_\pi$.

- Destrutor: Desaloca os objetos utilizados no processamento e imprime na saída padrão as estatísticas sobre a geração da função *hash*.

Classe Parser

Descrição: Efetua o *parsing* dos documentos da coleção, identificando as palavras e os limites entre os documentos.

Usa: NormHash

Função: Ler do disco o texto da coleção, identificando as palavras nele contidas e os limites entre documentos. Para cada palavra identificada, inserir a ocorrência na estrutura de dicionário especificada no construtor. Essa estrutura pode corresponder a um objeto da classe **NormHash** ou **PerfHash**, ambas descritas adiante. Quando um fim de documento é identificado, incrementar o contador de documentos. Coletar estatísticas sobre o texto da coleção.

Interface:

- Construtor: Aloca as estruturas internas ao objeto e inicializa as estatísticas a serem coletadas sobre o texto.

- *parse*(): Lê o texto do disco, efetuando o *parsing* de palavras e documentos.

- Destrutor: Libera as estruturas alocadas internamente.

Classe Option

Descrição: Processa a linha de comando do programa, configurando as opções de execução da forma especificada pelo usuário.

Função: Configurar as diversas opções de execução dos programas, a partir da linha de comando. As opções são especificadas no formato padrão do sistema UNIX, via $-\langle c \rangle$ ou $-\langle c \rangle\ \langle v \rangle$, onde c é um caractere especificador da opção, que pode ou não ser parametrizada, e v é o valor do parâmetro correspondente, se necessário.

Interface:

- *getOptions*(): Processa a linha de comando, configurando as opções mais comuns dos programas de geração do *hashing* e do índice.
- *getMachineList*(): Recebe como parâmetro uma *string* contendo nomes de máquinas separados por vírgulas e retorna uma lista com as máquinas a participarem da execução distribuída.
- *getDistribAlgo*(): Recebe uma *string* informando o nome do algoritmo distribuído a ser executado e retorna o código correspondente.

Classe NormHash

Descrição: Estrutura que implementa o tipo abstrato de dados dicionário, responsável pelo acesso às palavras do vocabulário durante a fase de geração da função *hash* perfeita. A estrutura implementada é uma tabela *hash* tradicional, com o emprego de vetores aleatórios para a função *hash* e *double hashing* para resolver colisões. Os diversos métodos para acesso ao vocabulário, com destaque para os vários tipos de tabelas *hash*, são discutidos na Seção 2.2. Maiores detalhes da implementação do objeto **NormHash** (*hashing* normal) são fornecidos na Seção 4.3.1.

Composta de: Vocab, RandVec

Usa: Graph

Função: Fornecer acesso eficiente às palavras do vocabulário da coleção (envolvendo inserção e busca), contabilizando o número de documentos em que cada termo ocorre. Ordenar o vocabulário da coleção e coordenar a geração da função *hash* perfeita para o mesmo. Para os algoritmos distribuídos, gerenciar o envio, recepção e intercalação das partes do vocabulário obtidas nas máquinas participantes.

Interface:

- Construtor: Aloca o corpo da tabela *hash* onde serão mapeados os termos do vocabulário, inicializa os parâmetros internos com as configurações recebidas e gera os objetos **Vocab** e **RandVec** usados para o armazenamento do vocabulário e o cálculo da função *hash*.
- *insert*(): Informa a ocorrência de um termo em um documento da coleção. O termo é associado a um inteiro (seu código *hash*) único e, caso seja sua primeira ocorrência na coleção, uma nova entrada para ele na estrutura da classe **Vocab** (descrita a seguir) é criada, correspondendo a sua inserção no vocabulário.
- *search*(): Busca por um termo no vocabulário da coleção, retornando o código associado.
- *sort*(): Ordena lexicograficamente os elementos do vocabulário da coleção.
- *writeVocab*(): Salva em disco o vocabulário ordenado da coleção. O arquivo gerado é composto pelas cadeias de caracteres correspondentes às palavras do vocabulário, precedidas por um byte indicando o tamanho da cadeia (no máximo, 255 caracteres).
- *sendVocab*(): Envia, para outra máquina participando da execução distribuída, o vocabulário produzido na máquina local. O formato do vocabulário enviado é o mesmo daquele gravado em disco.
- *recvVocab*(): Recebe o vocabulário produzido por outra máquina participando de um algoritmo distribuído.
- *updateVocab*(): Atualiza o vocabulário local com os elementos de outro recebido via rede. Este método é invocado quando uma máquina participante de uma geração distribuída recebe o vocabulário final produzido pela máquina de índice 0.
- *mergeVocab*(): Intercala o vocabulário recebido via rede com o vocabulário local. Invocado pelas máquinas situadas do lado esquerdo da atribuição na Subfase 3 do Algoritmo 3.1.
- *genPerfectHash*(): Gera a função *hash* perfeita para o vocabulário da coleção. Executada pelo algoritmo seqüencial ou pela máquina 0 de um algoritmo distribuído.
- *writePerfectHash*(): Escreve para disco a função *hash* obtida.
- *sendPerfectHash*(): Envia para as demais máquinas da rede a função *hash* gerada pela máquina 0 de uma execução distribuída.
- *recvPerfectHash*(): Recebe a função *hash* perfeita produzida pela máquina de índice 0, em uma execução distribuída.
- Destrutor: Libera a memória alocada para as estruturas internas e os objetos auxiliares.

Classe Vocab

Descrição: Armazena as cadeias de caracteres que compõem o vocabulário da coleção, isto é, o conjunto das palavras distintas que ocorrem no texto processado. Detalhes de sua implementação são descritos na Seção 4.3.2.

Função: Armazenar de forma eficiente e flexível as cadeias de caracteres correspondentes aos termos que formam o vocabulário da coleção. Permitir acesso direto a cada termo, sem gastos adicionais de espaço, e possibilitar o crescimento arbitrário do vocabulário.

Interface:

- Construtor: Aloca uma área para armazenamento do vocabulário baseado em uma estimativa do número de termos da coleção e inicializa as demais variáveis internas.
- *insert*(): Copia o conteúdo de um novo termo para a área de armazenamento da estrutura do vocabulário, redimensionando-a se necessário.
- *size*(): Retorna a posição atual do cursor de inserção na área de memória do vocabulário.
- *resize*(): Redimensiona a área de armazenamento destinada ao vocabulário.
- Destrutor: Libera a área de memória alocada.

Classe RandVec

Descrição: Proporciona um conjunto de valores inteiros aleatórios para serem usados na computação de uma função *hash* para cadeias de caracteres, como descrito na Seção 4.2.4.

Função: Gerar e armazenar um conjunto de inteiros aleatórios. Oferecer uma interface que permita seu uso no cálculo de uma função *hash* rápida e eficaz para cadeias de caracteres.

Interface:

- Construtor: Gera os inteiros aleatórios a serem utilizados no cômputo da função *hash*.
- *compute*(): Dada uma cadeia $k = k[1], \ldots, k[r]$, de r caracteres, utiliza o conjunto de inteiros aleatórios $a = a[1], \ldots, a[r_{\max}]$, gerado pelo construtor, para calcular uma função *hash* para k, dada por $h_a(k) = (\sum_{i=1}^{r} k[i] \times a[i]) \bmod m$, como descrito na Equação 2.2.
- *newData*(): Gera um novo conjunto de inteiros aleatórios para o objeto.

- *readData*(): Lê do disco um novo conjunto de inteiros aleatórios para o objeto.
- *writeData*(): Salva em disco o conjunto de inteiros do objeto.
- *sendData*(): Envia pela rede os inteiros aleatórios do objeto.
- *recvData*(): Recebe pela rede um novo conjunto de dados.
- Destrutor: Libera a memória utilizada para o armazenamento dos inteiros.

Classe Graph

Descrição: Implementação de grafos e hipergrafos através de listas de adjacências, com o objetivo de calcular uma função *hash* perfeita para um conjunto de chaves.

Composta de: Duas ou três instâncias de **RandVec**

Função: Dado um conjunto de chaves (no caso desta aplicação, o vocabulário da coleção), utilizar a técnica de grafos aleatórios, descrita em [20, 47], para gerar uma função *hash* perfeita para essas chaves. O algoritmo para a realização desse cálculo é descrito na Seção 2.2.2, enquanto que a Seção 4.3.3 mostra os detalhes de implementação.

Interface:

- Construtor: Aloca memória para as estruturas internas do grafo e instancia aos vetores aleatórios (**RandVec**) usados na computação da função.
- *createEdge*(): Dada uma cadeia de caracteres correspondente a um termo da coleção, cria uma aresta no grafo correspondente a esse termo.
- *deleteEdges*(): Remove todas as arestas do grafo, reinicializando-o para o cálculo de uma nova função *hash* perfeita.
- *cyclic*(): Determina se o grafo possui ou não um ciclo nas arestas geradas pelos termos da coleção. Caso afirmativo, ele não é adequado para o cálculo de uma função *hash* perfeita para o vocabulário.
- *assign*(): Dado que o grafo não é cíclico, rotula os vértices desse grafo, gerando assim a tabela de valores que permite a implementação da função *hash* perfeita.
- *restart*(): Caso o grafo seja cíclico, gera novamente os vetores aleatórios e reinicializa o grafo, sem suas arestas.

- *writeData*(): Escreve em disco os dados da tabela obtida na geração com sucesso de um grafo acíclico.
- *sendData*(): Envia pela rede os dados da função *hash* perfeita obtida.
- *recvData*(): Recebe pela rede uma função *hash* perfeita gerada com sucesso.
- Destrutor: Libera a memória alocada pelas estruturas internas e pelos objetos **RandVec**.

A Figura 4.2 apresenta um diagrama relacionando as principais classes da fase de indexação da coleção. A descrição dessas classes, suas funções e relacionamentos é feita a seguir. As seções seguintes trazem os detalhes de implementação das mesmas.

Classe DigIndex

Descrição: Controla o processo de geração do índice invertido para a coleção processada.

Usa: PerfHash, Parser, Option, Merge.

Cria: Proxy, RunIndex.

Função: Processar as opções para a geração especificadas pela linha de comando, estabelecer as conexões da máquina local com as demais (no caso de se executar um algoritmo distribuído), disparar o processamento do texto, a geração do índice intermediário e a intercalação para a produção do arquivo invertido final.

Interface:

- Construtor: Processa as opções de linha de comando, estabelece as conexões entre as máquinas participantes de um algoritmo distribuído, lê do disco arquivos gerados pela fase de geração da função *hash* perfeita e cria os principais objetos a serem utilizados no processamento.
- *parse*(): Realiza o *parsing* do texto, processando os arquivos da coleção e identificando palavras e documentos. Cada ocorrência de um termo em um documento é reportada para a classe **PerfHash**, descrita a seguir. Durante esta fase, são gerados em disco os *runs* ordenados, contendo as ocorrências identificadas, segundo o processo descrito na Seção 2.1.4.
- *merge*(): Executa a intercalação dos *runs* ordenados, gerados em disco durante o *parsing* do texto. Utiliza a técnica de intercalação por vários caminhos, citada na Seção 2.1.4, juntamente com a melhoria descrita na Seção 4.4.3, que permite a redução no número de *seeks* realizados a disco.

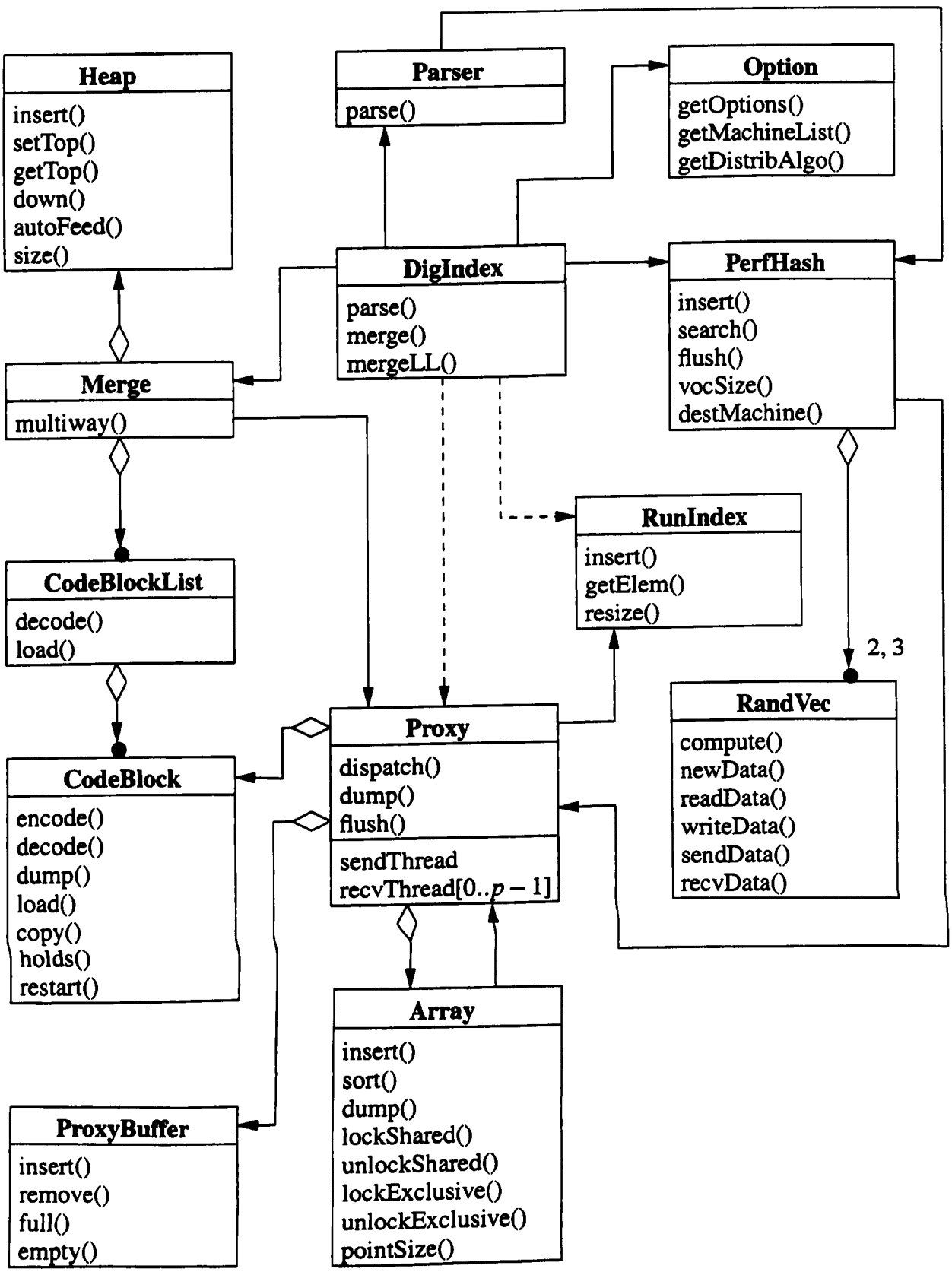


Figura 4.2: Diagrama de classes para fase de indexação.

- *mergeLL*(): Efetua a segunda intercalação dos arquivos invertidos locais, gerados por cada uma das máquinas participantes do algoritmo distribuído LL, de forma a produzir o arquivo invertido final.

- Destrutor: Desaloca os objetos utilizados no processamento e imprime na saída padrão as estatísticas sobre a geração do índice.

Classe PerfHash

Descrição: Tabela que implementa o tipo abstrato de dados dicionário, com o objetivo de oferecer acesso às palavras do vocabulário durante a fase de geração do índice. O método de acesso é o *hashing* perfeito, apresentado na Seção 2.2.2. Detalhes de implementação são fornecidos na Seção 4.4.1.

Composta de: Duas ou três instâncias de **RandVec**

Usa: **Proxy**

Função: Permitir acesso eficiente às palavras do vocabulário da coleção, de forma a se registrarem as ocorrências dos termos nos documentos. Armazenar as demais características do vocabulário como número de termos e, no caso dos algoritmos distribuídos, a máquina onde a lista invertida de cada termo será armazenada.

Interface:

- Construtor: Aloca memória para estruturas de dados e classes, lê do disco os dados da função *hash* perfeita, configurando as estruturas internas e os objetos **RandVec** com os dados lidos.

- *insert*(): Reporta a ocorrência de um termo do vocabulário em um documento. A classe **PerfHash** é responsável por contabilizar múltiplas ocorrências de um termo t_i em um mesmo documento d_j, identificando a freqüência relativa $f_{i,j}$ e inserindo o elemento $\langle t_i, f_{i,j}, d_j \rangle$ no arranjo de triplas, implementado pela classe **Array**, descrita adiante.

- *search*(): Retorna o identificador de um termo do vocabulário. Os termos são numerados a partir de 0, pela sua ordem lexicográfica.

- *flush*(): Descarrega as triplas $\langle t, f, d \rangle$ remanescentes nas estruturas da classe **PerfHash** e ainda não inseridas no arranjo.

- *vocSize*(): Retorna o tamanho do vocabulário da coleção, em número de termos.

- *destMachine*(): Gera um mapeamento que indica, para cada termo da coleção, qual máquina armazenará sua lista invertida ao fim da execução de um dos algoritmos distribuídos.
- Destrutor: Libera a memória alocada para as estruturas internas e os objetos **RandVec** utilizados no cálculo da função *hash*.

Classe Proxy

Descrição: Classe responsável por gerenciar a troca de triplas durante a geração do índice. Conhecedora com o algoritmo sendo executado (seqüencial, LL, LR ou RR), a classe **Proxy** seleciona o destino a ser dado às triplas identificadas durantes as etapas da indexação. Tais destinos podem corresponder ao arranjo de triplas, o disco ou a rede. **Proxy** tem por objetivo principal tornar transparente a troca de triplas pela rede na execução de um algoritmo distribuído. Na realização dessa tarefa, a classe **Proxy** conta com *threads* auxiliares: uma *sendThread*, para o envio de triplas pela rede, e $p-1$ *recvThreads*, para o recebimento de triplas das demais máquinas participando de uma indexação distribuída com p máquinas.

Composta de: CodeBlock, ProxyBuffer, Array

Usa: RunIndex

Função: Encapsular o tratamento das triplas à medida em que elas são identificadas. Há três instantes da indexação em que triplas são identificadas e cada algoritmo distribuído caracteriza-se por trocá-las pela rede em um desses instantes, como ilustrado nas Figuras 3.3, 3.5 e 3.7. **Proxy** também é responsável por disparar as *threads* auxiliares na execução de um algoritmo distribuído e por gerenciar os *buffers* de envio/recepção de triplas.

Interface:

- Construtor: Abre em disco os arquivos (temporário e final) onde as triplas destinadas à máquina local serão armazenadas, cria e conecta os *sockets* pelos quais as triplas "emigrantes" serão enviadas e as triplas "imigrantes" serão recebidas. Cria os blocos de código (objetos **CodeBlock**) usados para gravar as triplas em disco ou enviá-las pela rede. Dispara as *threads* responsáveis pelo envio e recepção das triplas.
- *dispatch*(): Direciona uma tripla identificada em um dado momento da indexação. Dada uma tripla $\langle t_i, f_{i,j}, d_j \rangle$, este método verifica qual deve ser seu destino, de acordo com os critérios descritos na Seção 4.5.2.

- *dump*(): Força o *dump* dos dados gerados no último *run* produzido. Isso envolve chamar o método *dump*() dos objetos **CodeBlock** associados e registrar a finalização do *run* na classe **RunIndex**, descrita adiante.

- *flush*(): Método invocado no momento em que a máquina termina o processamento de seu texto local. Para o algoritmo seqüencial e o LL, força a ordenação, compressão e gravação dos dados ainda armazenados no arranjo de triplas. Para os algoritmos LR e RR, os dados no arranjo de triplas são descarregados, enviados pela rede se necessário e as *threads* de envio e recepção são finalizadas.

- Destrutor: Libera a memória alocada para as estruturas de dados internas e invoca os destrutores dos objetos utilizados internamente à classe.

Processos:

- *main*: A *thread* principal (denominada aqui *main*) é responsável pela coordenação do processo de indexação, incluindo leitura e processamento do texto local. Em relação à classe **Proxy**, ela é responsável pela geração das triplas identificadas no texto local.

- *sendThread*: *Thread* responsável pelo envio das triplas "emigrantes" às máquinas de destino. Como descrito em maior detalhe na Seção 4.5.2, essa *thread* executa um laço infinito, procurando sempre remover as triplas inseridas no *buffer* de envio, representado pela classe **ProxyBuffer**. Sempre que uma tripla é removida, ela é inserida no bloco de código correspondente a sua máquina de destino. Esse bloco é descarregado para a rede ou para o disco quando se torna cheio.

- *recvThread*$[0, \ldots, p-1]$: Cada *thread* de recepção escuta o *socket* de conexão com uma das $p-1$ demais máquinas. Como detalhado na Seção 4.5.2, elas ficam bloqueadas esperando receber dados por seu *socket*. Quando se identifica um bloco de dados, as triplas são extraídas e inseridas no fluxo da máquina local.

Classe ProxyBuffer

Descrição: Depósito de triplas utilizado pela classe **Proxy** como área compartilhada para troca de dados entre a *thread* principal, que identifica as triplas do texto local, e a *sendThread*, que as envia pela rede. Sua estrutura de dados e implementação são baseadas no *buffer* do produtor/consumidor, um problema clássico em Ciência da Computação, encontrado em textos sobre sistemas operacionais e programação concorrente, como [79, 71, 77, 12]. A *thread* principal funciona como o produtor, gerando triplas e armazenando-as no *buffer*, enquanto que a *sendThread* faz o papel de consumidor, removendo itens do *buffer* e enviando-os pela rede.

Função: Prover uma área de memória compartilhada e os métodos de acesso a ela, de forma que a *thread* principal possa disponibilizar dados para a *thread* de envio de triplas sem ser bloqueada pela transmissão de dados pela rede. Da mesma forma, **ProxyBuffer** permite que a *sendThread* seja independente da *thread* principal, transmitindo as triplas de forma eficiente (em blocos) e permanecendo bloqueada enquanto uma quantidade suficiente de dados não está disponível. Os métodos de controle de acesso ao **ProxyBuffer** são baseados em técnicas de programação concorrente, descritas na Seção 4.5.2.

Interface:

- Construtor: Aloca a área de memória reservada para o *buffer* e inicializa as variáveis de controle de acesso.
- *insert*() Executada pela *thread* principal, insere uma tripla no *buffer*, caso este não se encontre cheio. Se o *buffer* está cheio, a *thread* principal fica bloqueada até que haja espaço disponível para a inserção. Por outro lado, se o *buffer* encontrava-se vazio antes da inserção, sinaliza-se para a *sendThread* indicando que há dados disponíveis.
- *remove*() Executada pela *thread* de envio, remove uma tripla do *buffer*, caso este não se encontre vazio. Se o *buffer* está vazio, a *thread* de envio fica bloqueada até que haja elementos disponíveis. Se, ao contrário, o *buffer* encontrava-se cheio quando a remoção foi realizada, a *thread* principal é sinalizada, indicando que há espaço disponível no *buffer*.
- *full*() Indica se o *buffer* encontra-se cheio.
- *empty*() Aponta se o *buffer* encontra-se vazio.
- Destrutor: Libera a memória alocada internamente para o *buffer*.

Classe RunIndex

Descrição: Índice que armazena as informações sobre os *runs* gerados ao longo da indexação.

Função: Concentrar as informações sobre cada *run* produzido pela indexação, de forma a possibilitar a intercalação por vários caminhos do índice temporário e a geração do arquivo invertido final.

Interface:

- Construtor: Aloca espaço para um índice de *runs*, dado um tamanho estimado. O número estimado de *runs* é dado por $R = \frac{x|\langle t,f,d\rangle|}{M}$, onde x é o número de triplas no texto local, $|\langle t,f,d\rangle|$ é o número de bytes ocupados por uma tripla em memória e M é a memória disponível para o arranjo de triplas, em bytes. Para o algoritmo LR, a estimativa corresponde ao valor acima multiplicado por p, o número de máquinas participando da execução.

- *insert*() Insere um elemento, correspondente a um novo *run*, no índice.
- *getElem*() Retorna o elemento localizado em uma dada posição no índice.
- *resize*() Redimensiona o índice, caso o tamanho estimado mostre-se menor que o real.
- Destrutor: Libera a memória alocada para o índice de *runs*.

Classe Array

Descrição: Implementa o *buffer* de triplas, apresentado na Seção 2.1.4 e responsável pelo armazenamento em memória dos pontos de indexação identificados. Provê um método para a inserção de triplas e para sua ordenação e gravação em disco, implementando assim as funcionalidades requeridas pela indexação seqüencial. Para o algoritmo RR, que depende do acesso concorrente de diversas *threads* ao *buffer* de triplas, **Array** oferece funções para exclusão mútua.

Usa: **Proxy**

Função: Prover armazenamento em memória principal para as triplas identificadas durante a indexação, implementando o *buffer B* descrito no Algoritmo 2.1. Oferecer métodos para a ordenação eficiente das triplas e sua gravação em disco.

Interface:

- Construtor: Aloca o bloco de memória principal onde serão armazenadas as triplas e inicializa as estruturas internas para o início das inserções.
- *insert*() Insere uma tripla no bloco de memória alocado. Caso este se esgote, a ordenação do arranjo e sua gravação em disco são disparadas.
- *sort*() Ordena as triplas inseridas no arranjo. A ordenação pode ser realizada através do tradicional *Quicksort* ou de outro método, específico para a ordenação de índices invertidos, descrito na Seção 4.4.2.
- *dump*() Grava as triplas armazenadas no arranjo, já ordenadas, em disco.
- *pointSize*() Retorna o número de bytes ocupados por uma tripla no arranjo. Como esclarecido na Seção 4.4.2, arranjos que implementam sua ordenação através do *Quicksort* podem consumir mais espaço por cada tripla, devido a restrições de alinhamento de memória.

  As demais funções abaixo são implementadas para suporte ao algoritmo RR e são invocadas pela *thread* principal e pelas *recvThread*s para operar sobre o arranjo. A descrição detalhada dessas operações encontra-se na Seção 4.5.2.

- *lockShared*() Garante à *thread* invocadora acesso compartilhado ao arranjo, utilizado para se inserir um novo elemento. Várias *threads* podem simultaneamente obter acesso compartilhado ao arranjo.
- *unlockShared*() Informa que uma *thread* já terminou a inserção de um elemento no arranjo.
- *lockExclusive*() Garante à *thread* invocando o método acesso exclusivo ao arranjo, permitindo-a ordená-lo e gravá-lo em disco. Apenas uma *thread* pode obter o acesso exclusivo ao arranjo em um dado momento.
- *unlockExclusive*() Libera a chave de acesso exclusivo obtida.
- Destrutor: Libera a memória alocada internamente para o *buffer* de triplas.

Classe Merge

Descrição: Classe que implementa o método de intercalação por vários caminhos, como apresentado na Seção 2.1.4.

Composta de: Heap, CodeBlockList

Usa: Proxy

Função: Realizar a intercalação eficiente dos *runs* de triplas gravados em disco ao longo do processamento do texto. Detalhes da implementação desta classe, incluindo como ela efetua a intercalação, são apresentados na Seção 4.4.4.

Interface:

- Construtor: Para uma intercalação de R caminhos, cria R listas de blocos de código utilizadas para a leitura dos dados do disco. Cria um *heap* de tamanho R para a ordenação das triplas, inicializando-o com um elemento de cada *run*.
- *multiway*() Efetua a intercalação por vários caminhos dos *runs* gerados durante o processamento do texto local. No caso do algoritmo LR, esses *runs* correspondem também a dados recebidos de outras máquinas da rede. Em relação ao algoritmo LL, essa função é responsável também pela intercalação final (por p caminhos) realizada para gerar o arquivo invertido final a partir dos trechos produzidos por cada uma das p máquinas executando o algoritmo.
- Destrutor: Libera a memória alocada para as listas de blocos de código

Classe Heap

Descrição: Classe que implementa o tipo abstrato fila de prioridades, através da estrutura de dados *heap* [69, 16]. **Heap** é usada para a ordenação das triplas presentes nos *runs* gerados durante o processamento do texto.

Função: Proporcionar uma estrutura de dados que permita a ordenação eficiente das triplas presentes nos R *runs* obtidos. De forma a suportar essa operação, um *heap* é uma estrutura com R elementos $p[1], p[2], \ldots, p[R]$, respeitando uma propriedade de ordenação que estabelece que $p[i] \leq p[2i]$ e $p[i] \leq p[2i+1]$, para todo $i = 1, 2, R/2$. Esta ordenação pode ser visualizada imaginando-se a seqüência de elementos como uma árvore binária, onde os filhos de um elemento de índice i correspondem aos elementos nas posições $2i$ e $2i+1$.

Interface:

- Construtor: Aloca memória para $R+1$ elementos, inserindo uma sentinela na posição 0.
- *insert*() Insere um novo elemento ao fim do *heap*, subindo-o até a posição que mantenha sua propriedade de ordenação.
- *setTop*() Copia um novo elemento para a primeira posição do *heap* (a de índice 1).
- *getTop*() Retorna o elemento presente na posição 1 do *heap*.
- *down*() Desloca o elemento a uma dada posição do *heap*, movendo-o para posições inferiores até que a propriedade de ordenação seja reestabelecida.
- *size*() Retorna o número de elementos presentes no *heap*.
- *autoFeed*() Substitui o primeiro elemento do *heap* pelo último, decrementando de um o tamanho do *heap*. Utilizado quando não há mais elementos disponíveis nos *runs*.
- Destrutor: Libera a memória alocada para os elementos do *heap*.

Classe CodeBlock

Descrição: Implementa um bloco para codificação/decodificação, gravação/leitura em disco e transmissão via rede de triplas.

Função: Proporcionar uma estrutura para o armazenamento das triplas identificadas durante a indexação. Deve ser possível especificar se as triplas serão mantidas comprimidas ou não. Prover operações eficientes para leitura/escrita de triplas em dispositivos, utilizando para isso blocos de código, como detalhado na Seção 4.4.3.

Interface:

- Construtor: Aloca memória para o bloco onde serão armazenados os códigos das triplas. Configura o objeto, indicando se as triplas devem ser armazenadas comprimidas ou não, se o bloco será usado para codificação ou decodificação de dados e qual o dispositivo onde leituras ou escritas devem ser executadas.
- *encode*() Codifica uma tripla, copiando seu código para o bloco.
- *decode*() Decodifica a tripla na posição corrente do bloco.
- *dump*() Escreve o bloco de código para o dispositivo configurado na criação.
- *load*() Carrega um novo bloco a ser decodificado do dispositivo configurado.
- *copy*() Copia o conteúdo de um bloco de código para outro.
- *holds*() Indica se um bloco contém espaço livre suficiente para o armazenamento do código de uma tripla.
- *restart*() Reinicializa um bloco para a inserção de novas triplas (no caso de um bloco de escrita) ou decodificação de dados (no caso de um bloco de leitura).
- Destrutor: Libera a memória alocada para o bloco de código.

Classe CodeBlockList

Descrição: Lista de blocos de código utilizada durante a intercalação dos *runs* para a redução do número de *seeks* a disco.

Composta de: CodeBlock

Função: Proporcionar uma diminuição no número de *seeks* a disco efetuados durante a intercalação dos *runs*, pré-carregando um conjunto de blocos de código para a memória, como descrito na Seção 4.4.4. Propagar as operações executadas sobre a lista – apenas *decode*() e *load*(), já que os blocos são utilizados sempre para decodificação – sobre o bloco de código ativo, passando ao próximo bloco da lista quando o bloco corrente se esgota e carregando um novo conjunto de blocos somente quando a lista chega ao fim.

Interface:

- Construtor: Aloca memória para uma lista de objetos da classe **CodeBlock**, inicializando cada elemento da lista.
- *decode*() Decodifica a tripla na posição corrente do bloco ativo da lista.
- *load*() Carrega um novo conjunto de blocos para a memória.
- Destrutor: Libera a memória alocada para cada bloco de código presente na lista.

4.2 Classes de Suporte

Nesta seção descrevem-se as classes implementadas para oferecer suporte às fases de obtenção do vocabulário e geração do índice invertido. Algumas delas não foram relacionadas nos diagramas de classes apresentados na Seção 4.1 por serem de uso comum a todas as demais classes e proverem funções básicas, não específicas ao sistema **DIG**.

4.2.1 Tratamento de erros e chamadas de sistema

O sistema **DIG** utiliza um conjunto de rotinas específicas para relato e tratamento de erros, como sugerido em [75]. Foi criada uma série de funções com uma assinatura comum: todas recebem como parâmetro uma *string* de formato e um conjunto de argumentos, assim como a função `printf` da biblioteca padrão C. Cada uma, entretanto, trata o erro de uma forma distinta: somente imprimindo a mensagem especificada, abortando a execução ou ainda gerando um *core dump*. Dessa forma, erros de diversas gravidades podem ser tratados da forma apropriada, através de uma simples chamada de função.

Entretanto, tratar devidamente os erros em todos os trechos do código onde eles podem ocorrer é uma tarefa tediosa, que aumenta significativamente o volume de código produzido e tende a não ser cumprida com rigor na ausência de instrumentos adequados para auxiliá-la. Linguagens como Java [4] e C++ [78] procuram oferecer suporte ao tratamento de erros através do mecanismo de **exceções**.

No caso do sistema **DIG**, elaborado em linguagem C, esse suporte nativo não existe. Criou-se para ele então um conjunto de funções de encapsulamento (como as *wrapper functions*, sugeridas em [75, 76]), que englobam as chamadas a diversas chamadas de sistema (*system calls*) e outras funções da biblioteca padrão C e da biblioteca de *threads* POSIX. Tais funções de encapsulamento verificam os valores de retorno das chamadas que executam e podem tratar erros apropriadamente.

Por outro lado, o tratamento dos erros pode variar de acordo com o contexto. Uma escrita sem sucesso no arquivo contendo o índice invertido final deve gerar um erro fatal, mas um erro

em uma tentativa de escrita em um *socket* de rede deve gerar uma mensagem de aviso e ser refeita mais adiante. Implementou-se, portanto, para cada função encapsulada, um *handler*, representado por um ponteiro para a função de tratamento de erro a ser invocada quando um comportamento anormal é observado na chamada da função. O *handler* pode ser alterado em um determinado trecho da execução para, por exemplo, somente imprimir uma mensagem de erro e em outro para abortar o programa.

4.2.2 Tratamento de arquivos, opções de comando e outras funções

O sistema **DIG** faz uso uma classe utilitária para a manipulação de arquivos em disco. Além de abertura, fechamento e verificação de propriedades de arquivos, essa classe implementa uma varredura recursiva de todos as entradas a partir de um determinado ponto do sistema de diretórios. Tal funcionalidade é utilizada para se permitir a indexação completa de uma árvore de diretórios contendo arquivos de texto, como é a coleção TREC. Esta, em sua versão 3, traz o texto dividido em três diretórios, um para cada disco da distribuição, cada qual por sua vez contendo um subdiretório para cada subcoleção (como *Wall Street Journal*, *Ziff Publishing* e outras).

A classe **Option**, descrita na página 60, implementa o processamento das opções de linha de comando, permitindo que o sistema **DIG** aceite uma série de parâmetros que configuram sua execução. Tais parâmetros são especificados no formato $-\langle c \rangle$ ou $-\langle c \rangle\ \langle v \rangle$, onde c é um caractere especificador da opção, que pode ou não ser parametrizada, e v é o valor do parâmetro correspondente, se necessário. Eles permitem o controle de opções como quais máquinas serão utilizadas em um algoritmo distribuído, quantos bytes devem ser reservados em memória para o armazenamento de triplas ou ainda se os blocos de código devem ou não ser comprimidos, como mostrado na Seção 5.2.

Desenvolveu-se também uma classe contendo várias funções utilitárias que não se enquadrariam em nenhuma das demais classes do sistema. Entre essas incluem-se funções matemáticas, como para o cálculo do número primo mais próximo de um dado valor, e de levantamento dos parâmetros do sistema, como o cálculo da memória principal livre em um sistema Linux.

4.2.3 *Parsing* da entrada

Uma importante tarefa executada tanto na fase de obtenção do vocabulário como na de geração do índice é o *parsing* do texto. Como será mostrado no Capítulo 5, uma parte significativa do tempo de execução de ambas as fases é gasta processando o texto da coleção, identificando palavras e documentos. Essa é uma responsabilidade da classe **Parser**, cuja interface foi apresentada na página 60 e para a qual a implementação é descrita a seguir.

As palavras consideradas pelo sistema **DIG** são compostas por caracteres alfanuméricos contendo no máximo quatro dígitos e 256 no total caracteres por palavra, segundo o critério

Classe	Máscara
Separadores	00001
Letras	00010
Dígitos	00100
Início de *tag* ('<')	01000
Fim de *tag* ('>')	10000

Tabela 4.1: Classes de caracteres do *parsing* e suas máscaras correspondentes.

estabelecido na Seção 2.1.2. De forma a identificar de forma eficiente essas palavras, bem como as marcas separadoras de documentos, **Parser** utiliza máscaras de bits, o que lhe permite estabelecer classes de caracteres.

Uma tabela interna a um objeto **Parser** define uma classe ou categoria para cada um dos 256 caracteres do alfabeto. Para o *parsing* do texto da coleção TREC (em SGML), há cinco classes básicas, mostradas na Tabela 4.1, sendo cada uma correspondente a uma máscara de bits.

Máscaras derivadas são criadas para definir outros itens, de acordo com os critérios estabelecidos. Palavras são representadas pela máscara 00110 (letras ou dígitos), enquanto que os limites de *tags* podem ser identificados pela máscara 11000. A identificação de palavras e documentos pode então ser feita de forma muito mais rápida do que quando se utilizam as funções da biblioteca C como `isspace`, `isalnum` e `isdigit`. O texto é lido do disco em blocos, que são percorridos caractere a caractere, realizando-se apenas operações sobre bits, bastante eficientes.

Quando caracteres alfanuméricos são identificados, eles são convertidos para minúsculo e têm sua acentuação removida, sendo então copiados para uma outra área de memória, que conterá a palavra do vocabulário. Essa conversão utiliza também uma tabela estática interna à classe **Parser**. Quando uma marca de final de documento (para a TREC, "`</DOC>`") é identificada, o índice do documento corrente é incrementado. Deve-se dar atenção aos casos em que uma palavra ou uma *tag* encontra-se quebrada entre dois blocos do texto.

4.2.4 Vetores aleatórios

Os vetores aleatórios, forma pela qual convencionou-se chamar os arranjos contendo inteiros gerados aleatoriamente para a computação de funções *hash* para *strings*, desempenham papel fundamental no acesso ao vocabulário do texto em ambas as fases do processamento. Isso porque, tanto a tabela *hash* normal empregada na primeira fase como a tabela *hash* perfeita utilizada na fase final, são dependentes dessa estrutura para obter os índices dos termos do vocabulário. A teoria sobre a implementação de tabelas *hash* normais e perfeitas foi discutida nas Seções 2.2.1 e 2.2.2, respectivamente, enquanto que a forma pela qual essas tabelas usam os vetores aleatórios para implementar esse acesso é descrita nas Seções 4.3.1 e 4.4.1.

Assim como ocorre com a classe responsável pelo *parsing*, a eficiência dos vetores aleató-

rios tem impacto direto sobre o tempo total de execução da indexação. Isso é devido ao grande número de palavras que pode haver em um texto; por exemplo, um texto de 3 gigabytes terá aproximadamente 500 milhões de palavras. Nesta seção, descreve-se como se implementa a computação eficiente de funções *hash* para palavras a partir de vetores aleatórios.

Objetos da classe **RandVec**, responsável pela implementação dos vetores aleatórios, são instanciados especificando-se um limite dentro do qual deve estar o valor da função *hash* computada para uma dada *string*. Esse valor corresponde ao tamanho m da tabela *hash* onde as *strings* serão mapeadas, de forma que todos os valores computados devem estar no intervalo $0,\ldots,m-1$. Para garantir isso, usa-se normalmente a função módulo. Observou-se, entretanto, que essa função, por realizar uma divisão inteira, é bastante lenta quando comparada com outras operações aritméticas, como soma e subtração. No código, substituiu-se essa função por um teste, que verifica, a cada novo termo $k[i] \times a[i]$ adicionado, se o valor $h(k)$ sendo computado excede o limite m. Caso afirmativo, toma-se o valor $h(k) - m$ para a próxima iteração do somatório.

No construtor da classe é gerada também a massa aleatória de dados a ser utilizada na computação. Como citado na Seção 2.2.1, a operação de multiplicação da Equação 2.2 é evitada gerando-se um valor aleatório distinto para cada um dos 256 caracteres distintos do alfabeto. Considerando que o tamanho máximo de uma palavra é também de 256 caracteres, uma massa de dados completa conteria $256 \times 256 \times 4 = 256$ Kbytes (considerando inteiros de 4 bytes). De forma a reduzir esse custo, criam-se valores aleatórios distintos para apenas os 16 primeiros caracteres das *strings*, repetindo-os para os caracteres seguintes. Essa modificação não tem impacto significativo no espalhamento proporcionado pela função *hash*, já que a maioria das palavras possui menos que 16 caracteres.

A eficácia no espalhamento das chaves proporcionado pelas funções *hash* obtidas através de vetores aleatórios está fundamentada na teoria de *hashing* universal, apresentada na Seção 2.2.1 e suportada por referências como [42, 16]. Experimentos realizados em estágios precoces do desenvolvimento do sistema **DIG** mostraram ser essa abordagem muito superior em termos de tempo de computação e espalhamento de chaves quando comparada com outras técnicas populares para implementação de funções *hash*. Entre estas últimas, pode-se citar o uso de um valor primo como fator multiplicador para os caracteres das *strings* (como sugerido em [69]) ou o emprego da função de cálculo do CRC (*Cyclic Redundancy Check*).

4.3 Obtenção do Vocabulário

Nesta seção descrevem-se os detalhes de implementação das principais classes envolvidas na fase de obtenção do vocabulário. Focalizam-se agora apenas os aspectos ligados ao processamento seqüencial; a implementação específica do levantamento do vocabulário global de uma coleção distribuída será discutida na Seção 4.5.1.

4.3.1 *Hashing* normal

A tabela *hash* normal empregada para o levantamento do vocabulário, implementada pela classe **NormHash**, utiliza, como já citado, vetores aleatórios para a implementação de sua função de espalhamento. Na criação da tabela é especificada uma estimativa v_{est} do número de chaves a serem nela inseridas. Essa estimativa é usada para determinar o tamanho inicial da tabela e controlar sua taxa de ocupação, expandindo-a de forma a evitar colisões excessivas. Ela deve ser calculada a partir do tamanho do texto e de algum modelo que aponte o número aproximado de palavras no vocabulário. Para a coleção TREC, esse modelo foi levantado em [2] e diz que, dado um texto de n bytes, o número de palavras no vocabulário pode ser estimado por $v_{est} = 4.8 \times n^{0.56}$. Caso não se disponha de um modelo como esse ou não se saiba antecipadamente o tamanho do texto, uma estimativa qualquer deve ser fornecida, sob o risco de ser necessário reconstruir a tabela com novo tamanho.

A taxa de ocupação da tabela, conhecida por fator de carga e representada pelo símbolo α, é sempre mantida entre dois limites, um inferior, α_{min}, e um superior, α_{max}, também especificados na construção do objeto **NormHash**. Inicialmente, a tabela é criada com um número de vagas $m = v_{est}/\alpha_{max}$, de forma que, caso a estimativa do número de chaves seja correta, a ocupação máxima somente seja atingida ao final do processamento e não se faça necessária a reconstrução da tabela. À medida que novos termos são inseridos, verifica-se o número de termos v e, caso ele ultrapasse a estimativa v_{est}, uma nova tabela é construída contendo $m = v/\alpha_{min}$ posições e recebe os elementos presentes na tabela antiga, que é então destruída. O sistema **DIG** usa os valores 0.25 para α_{min} e 0.75 para α_{max} pelo fato desse intervalo proporcionar um reduzido número de colisões, como ilustrado pelo gráfico da Figura 5.5. Com sua taxa média de ocupação – 0.5 – a tabela *hash* incorre em apenas uma colisão por acesso.

Ainda na construção da tabela é criado também o vetor aleatório usado internamente no cálculo da função *hash* para os termos da coleção. Esse vetor é parametrizado com o tamanho m da tabela e deve, portanto, ser novamente gerado a cada vez que a tabela muda de tamanho. Como descrito, devido aos custos envolvidos na reconstrução da tabela *hash* quando ela se torna saturada, deve-se evitar essa operação, estimando sempre que possível o número de termos na coleção.

Uma vez criada a tabela, o texto é processado e cada palavra identificada tem sua ocorrência reportada através da função *insert*(). O índice para o qual a palavra deve ser mapeada na tabela é determinado através da função *search*(). Dado esse índice, verifica-se se a ocorrência do termo no documento atual ainda não foi reportada. Nesse caso, indica-se que a última ocorrência do termo foi no documento atual e incrementa-se a freqüência f_t, que contabiliza o número de documentos da coleção onde o termo ocorre. Se esta também é a primeira vez que o termo foi encontrado na coleção, ele é copiado para a estrutura que armazena o vocabulário.

A função *search*(), por sua vez, é responsável por computar em qual posição da tabela cada termo será mapeado. Para isso, utiliza-se em primeiro lugar o valor da função *hash* computada

pelo vetor aleatório interno ao objeto **NormHash**. Caso esse valor gere uma colisão, isto é, caso a posição da tabela para o qual o termo foi mapeado já esteja ocupada por um termo diferente, utiliza-se uma segunda função *hash* para a resolução, segundo a filosofia de *double hashing* [42, 16]. Para proporcionar eficácia ao cálculo de uma nova posição de inserção, essa nova função deve, ao mesmo tempo, ser de computação rápida e proporcionar, assim como a primeira função, um espalhamento aleatório das palavras.

Para isso usa-se uma nova massa de dados aleatórios, interna à classe **NormHash**, que mapeia cada um dos 256 caracteres do alfabeto em um inteiro primo entre 3 e $m/2+3$. Esse intervalo de valores é reputado por oferecer bons resultados como uma segunda função *hash*, de acordo com [69]. Uma maior dependência da segunda função *hash* em relação à palavra sendo processada é obtida endereçando-se a nova massa de dados com o primeiro caractere da palavra. Essa nova variável no cálculo evita um fenômeno comum no uso de *double hashing* com a segunda função fixa, chamado *clustering*, que corresponde ao acúmulo de colisões sobre as posições da tabela distantes umas das outras exatamente pelo valor da segunda função *hash*. O Algoritmo 4.1 formaliza o cálculo do índice i de um termo k na tabela T. O valor c da segunda função *hash* é seguidamente subtraído do valor da primeira, corrigindo-se o resultado caso ele se torne inferior a zero. Isso evita o uso da cara função módulo para manter i entre 0 e $m-1$.

Entradas: A *string* k, **NormHash** T, **RandVec** $T.r$ e o vetor de inteiros aleatórios $T.c$
Saídas: i, o índice de k em T

```
i ← T.r.compute(k)
se T[i] = ∅ ∧ T[i].k = k então
  return i
senão
  c ← T.c[k[0]]
  i ← i − c
  se i < 0 então
    i ← i + T.m
  fim se
  enquanto T[i] ≠ ∅ ∨ T[i].k ≠ k faça
    i ← i − c
    se i < 0 então
      i ← i + T.m
    fim se
  fim enquanto
  return i
fim se
```

Algoritmo 4.1: Cálculo da posição de inserção de uma chave k em uma tabela *hash* comum T.

4.3.2 Armazenamento do vocabulário

Enquanto que o acesso aos termos do vocabulário é proporcionado pela tabela *hash*, as *strings* correspondentes a esses termos devem ser armazenadas adequadamente e mantidas acessíveis. Isso deve ser feito para que se possam realizar as comparações que permitem a identificação de colisões e para que, terminado o processamento do texto, seja possível gerar a função *hash* perfeita para o vocabulário da coleção. Tal função é desempenhada pela classe **Vocab**, descrita na página 60.

Um objeto **Vocab** é construído especificando-se uma estimativa para o número de termos do vocabulário, o mesmo valor passado para a criação da tabela *hash*. Internamente, calcula-se o número de bytes a serem ocupados pelos caracteres desses termos, utilizando-se o tamanho médio das palavras distintas de um texto, o parâmetro $|w_V|$ introduzido na Tabela 1.1. Para textos na maioria das línguas ocidentais, esse valor fica em torno de 8 caracteres por palavra.

A cada novo termo identificado no texto, seus caracteres são copiados para um arranjo de bytes interno ao objeto **Vocab**, precedidos por um byte indicando o tamanho da cadeia. Neste ponto, tira-se vantagem do limite de 256 para o número de caracteres por termo – os indicadores de tamanho podem ser armazenados em um único byte, seguidos pela *string* correspondente. A referência para os caracteres de um termo é feita através do índice do arranjo de bytes onde se encontra seu indicador de tamanho. Caso a inserção de um novo termo exceda o espaço disponível no arranjo, este é realocado com 50% de memória a mais que a capacidade atual. Tal ajuste é feito, portanto, de forma independente do controle de carga da tabela *hash*.

4.3.3 Grafos

Terminado o processamento do texto e ordenado o vocabulário, passa-se à geração da função *hash* perfeita para o conjunto dos termos identificados. Nessa tarefa, o sistema **DIG** emprega hipergrafos aleatórios, propostos nos trabalhos de Czech *et al.* [20, 36, 47, 19], cuja teoria foi exposta na Seção 2.2.2. A seguir discute-se a implementação de grafos que provê suporte à construção de uma função *hash* perfeita ou, segundo a sigla em inglês, uma OPMPHF (de *Order Preserving Minimal Perfect Hash Function*).

As operações sobre grafos necessárias para a geração de OPMPHFs são providas pela classe **Graph** que, por sua vez, utiliza listas de adjacências como estrutura de dados interna. Entretanto, uma implementação ingênua de grafos através de listas de adjacências, da forma descrita em livros básicos de algoritmos, pode levar a ineficiências em termos de uso de memória e tempo de execução, devido ao grande número de ponteiros e à excessiva alocação dinâmica de pequenas quantidades de memória. Quando se trata de hipergrafos, cujas arestas ligam três ou mais vértices simultaneamente, esse tipo de implementação torna-se impraticável.

Para representar um grafo comum de m vértices e n arestas, uma implementação tradicional por listas de adjacências alocaria inicialmente um vetor de m listas, correspondendo aos vértices

do grafo. Considerando que cada lista é implementada por uma estrutura de 8 bytes (um ponteiro de 4 bytes para o início da lista e outro para o fim) e que cada vértice possui também um rótulo (um inteiro de 4 bytes), teriam sido alocados já $12m$ bytes. As n arestas seriam representadas por $2n$ elementos nas listas, cada um pertencente à lista de um dos dois vértices conectados. Cada elemento seria composto por também um rótulo (outro inteiro de 4 bytes), um inteiro de 4 bytes apontando para o vértice com o qual a conexão é feita e um ponteiro de 4 bytes para o próximo elemento da lista, o que consumiria mais $24n$ bytes. Considerando-se um vocabulário de $n = 1.000.000$ elementos e escolhendo-se $m = 2.5n$, como no exemplo da Seção 2.2.2, seriam consumidos aproximadamente 54 Mbytes para o grafo, sendo realizadas 2.000.000 chamadas à função de alocação dinâmica `malloc()`, o que gera uma ocupação de memória ainda maior com metainformação sobre os dados alocados.

Para hipergrafos, o consumo seria ainda maior. Considerando-se trigrafos, cada aresta implicaria na criação de três elementos, um na lista de cada um dos três vértices ligados. Cada aresta deveria conter não somente um, mas dois inteiros apontando para os outros vértices conectados através dela. Isso implicaria num custo de 48 bytes por aresta, levando a um consumo impraticável de aproximadamente 78 Mbytes para a situação com um vocabulário de 1.000.000 elementos. Considerando-se grafos de mais dimensões, o consumo cresce ainda mais. A solução é utilizar blocos pré-alocados de memória para armazenar os vértices do grafo e os ponteiros correspondentes às arestas.

A implementação de grafos utilizada no sistema **DIG** emprega três arranjos, cada um contendo elementos alocados em posições contíguas de memória. Um primeiro arranjo, denominado *Edges*, possui n elementos, cada um correspondendo a um termo do vocabulário. Cada elemento desse arranjo consiste em d inteiros, onde d é a dimensão do grafo (2 para grafos comuns, 3 para trigrafos, e assim em diante) e, indica, para cada aresta e, os vértices por ela conectados. Outro arranjo, *First*, possui m elementos e aponta, para cada vértice v, a primeira aresta e a incidir sobre v. O terceiro arranjo, *Next*, possui $n \times d$ elementos e aponta, para cada aresta, o próximo vértice sobre o qual ela incide. Essa implementação leva a um custo de cerca de 26 Mbytes para grafos bidimensionais quando $n = 1.000.000$, sem a sobrecarga causada pelo grande número de itens alocados dinamicamente. Para trigrafos, a ocupação seria de 34 Mbytes, menos que a metade dos 78 Mbytes alocados pela implementação trivial.

Geração do *hashing* perfeito

O grafo é construído pela inserção sucessiva de suas arestas, correspondentes aos termos da coleção. Estando o vocabulário ordenado lexicograficamente, para cada termo i, $0 \leq i \leq v-1$, aplicam-se as funções *hash* proporcionadas pelos d vetores aleatórios internos ao grafo. Com isso, obtêm-se uma tupla de d inteiros $(h_0, h_1, \ldots, h_{d-1})$. Verifica-se então se essa tupla não gera um laço no grafo, o que ocorre caso existam h_i e h_j, com $i \neq j$, tais que $h_i = h_j$. Nesse caso, um grafo cíclico seria gerado, o que impede a obtenção de uma função *hash* perfeita para o dado

Índice	Termo	Tupla (h_0, h_1)
0	a	(4,8)
1	ainda	(2,8)
2	anda	(4,5)
3	aonde	(10,11)
4	onda	(3,12)

Tabela 4.2: Tuplas de funções *hash* produzidas para o vocabulário da coleção-exemplo.

conjunto de chaves e obriga o grafo a ser reinicializado, gerando-se novos vetores aleatórios.

Caso a d-tupla não gere um laço, ela é inserida na i-ésima posição do arranjo *Edges* e, para cada h_j, $0 \leq j \leq d-1$, atribui-se o valor em $First[h_j]$ a $Next[i + j \times m]$ (significando que a aresta i incide também sobre o vértice h_j) e assinala-se i a $First[h_j]$ (indicando que a primeira aresta incidindo sobre o vértice h_j agora é i). Exibe-se a seguir um exemplo de criação de um grafo aleatório de duas dimensões, utilizando-se para isso o vocabulário da coleção-exemplo mostrada na Tabela 2.1. A Tabela 4.2 mostra uma conjunto de funções *hash* hipotéticas geradas para cada termo pelos dois vetores aleatórios do grafo. Para esse exemplo, tem-se um vocabulário de 5 termos ($n = 5$) e toma-se m como o valor primo mais próximo de $2.5n$ (resultando em $m = 13$).

A Figura 4.3 traz a representação visual do grafo após as inserções das arestas, enquanto que a Figura 4.4 mostra como ficariam os arranjos da implementação do grafo ao fim da criação. As posições marcadas com uma barra não foram inicializadas. O vetor *Next* é dividido em duas metades, cada uma relativa a uma dimensão do grafo.

Uma vez construído o grafo, a tarefa de geração da função *hash* perfeita não está concluída, pois, segundo a teoria, ele não será capaz de gerar o mapeamento dos termos nos seus identificadores caso possua ciclos. A verificação deve ser feita e, caso seja detectado um ciclo, novos vetores aleatórios devem ser gerados e um novo grafo, construído. A detecção de ciclos é implementada pela função *cyclic*() da classe **Graph**, que se baseia na propriedade de grafos e hipergrafos citada na Seção 2.2.2. Essa propriedade estabelece que, se forem removidas todas as arestas de um grafo que incidem apenas sobre vértices de grau 1 (isto é, vértices ligados a apenas uma aresta), então esse grafo é acíclico se e somente se o grafo resultante não contiver nenhuma aresta.

Portanto, a função *cyclic*() primeiro verifica se, para cada vértice v, $0 \leq v \leq m-1$, $First[v]$ é um valor válido, isto é, se há alguma aresta e incidente a v. Caso afirmativo, é observado se $Next[e]$ é vazio, indicando que e é a única aresta sobre v e este é um vértice de grau 1. Se verdadeito, a aresta e é então removida do grafo, eliminando-se os elementos na lista encadeada simulada por *Next*. À medida que as arestas são removidas, elas são inseridas em uma pilha S. Quando não houver mais vértices de grau 1 no grafo e, portanto, não for possível remover mais arestas, verifica-se o número de elementos na pilha S. Se esse número for igual a n,

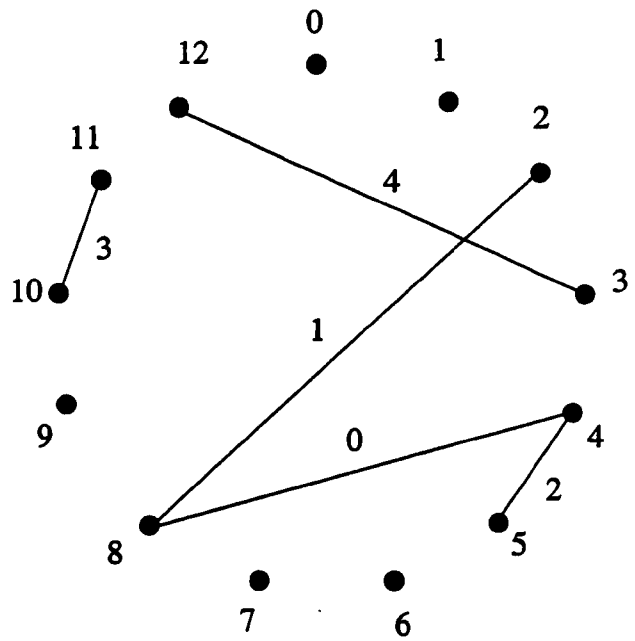


Figura 4.3: Representação visual para o grafo criado sobre os termos da coleção-exemplo.

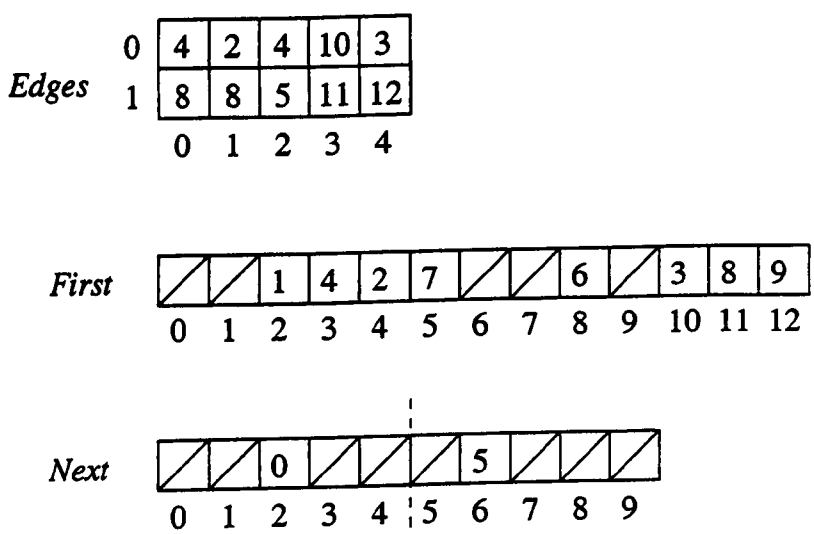


Figura 4.4: Configuração dos vetores de implementação do grafo.

então todas as arestas foram removidas e o grafo é acíclico. Caso contrário, há ciclos e um novo grafo aleatório deve ser gerado. Quando for obtido um grafo acíclico, fato que ocorrerá em um número finito de tentativas caso a proporção entre n e m seja mantida da forma descrita na Seção 2.2.2, os vértices do grafo são rotulados da forma descrita no Algoritmo 2.3. Gera-se então o mapeamento g que permite, juntamente com os vetores aleatórios utilizados, a implementação da função *hash* perfeita.

4.4 Geração do Índice

Apresentam-se agora mais informações sobre a implementação sobre as classes envolvidas na segunda fase do processamento do texto, a geração propriamente dita do índice invertido. Nesta seção abordam-se os aspectos da geração seqüencial, enquanto que a indexação distribuída será discutida na seção seguinte.

4.4.1 *Hashing* perfeito

Na fase de geração do índice, o acesso ao vocabulário é provido pela tabela *hash* perfeita, utilizando a função gerada como resultado da primeira fase. A tabela é construída especificando-se o nome do arquivo onde foram gravados os dados da função *hash* perfeita. Lê-se desse arquivo as informações sobre o tamanho v do vocabulário, número de elementos m do mapeamento g e o número de dimensões d do grafo que gerou a função. Os dados dos d vetores aleatórios são então lidos, assim como os m inteiros do vetor g. A tabela é alocada e seus n elementos, inicializados.

Essa tabela opera da seguinte forma. Assim como é feito com a tabela *hash* normal, para cada termo t_i identificado no documento d_{cur} sendo processado, sua ocorrência é reportada através do método *insert*(), que, por sua vez, invoca *search*() para determinar o índice i sob o qual o termo será mapeado. Uma vez obtido esse índice, verifica-se se a ocorrência do termo no documento atual já foi reportada. Caso afirmativo, incrementa-se a freqüência relativa $f_{i,\text{cur}}$. Senão, e se essa não for a primeira ocorrência do termo na coleção, toma-se sua última ocorrência (diga-se, em um documento d_j) e produz-se a tripla $\langle t_i, f_{i,j}, d_j \rangle$, que é despachada para o objeto **Proxy**, responsável pelo gerenciamento das triplas.

Por sua vez, o método *search*() calcula o índice i através dos vetores aleatórios e do mapeamento g internos ao objeto **PerfHash**. Como introduzido na Seção 2.2.2, dada a *string* k correspondente ao termo, esse índice é calculado como

$$i = \left[g \left(\sum_{j=0}^{d-1} h_j(k) \right) \right] \bmod n,$$

onde d é o número de dimensões do grafo usado na geração da função *hash* e h_j são as funções *hash* proporcionadas pelos d vetores aleatórios. O sistema **DIG** permite apenas o uso de grafos de duas ou três dimensões.

4.4.2 Arranjo de triplas

O arranjo de triplas, implementado pela classe **Array** (página 71), corresponde à área de memória onde as triplas são armazenadas. Um objeto é instanciado especificando-se sua capacidade de armazenamento em número de triplas, o que é calculado dividindo-se a memória disponível para a indexação pelo espaço ocupado por uma tripla. Na instanciação do arranjo informa-se também se ele deverá ordenar seus elementos através do *Quicksort* (utilizando a função C `qsort()`) ou através de um método desenvolvido para explorar a distribuição das freqüências em listas invertidas. Esse parâmetro determinará a estrutura interna do arranjo, que difere significativamente de uma opção para a outra, como será visto adiante.

A função do arranjo é simples, como se vê pelo Algoritmo 2.1: armazenar as triplas identificadas no texto local, ordenando-as quando necessário e comprimindo-as, se assim especificado. Na realidade, após serem ordenadas, as triplas são despachadas para o objeto **Proxy**, que pode optar por gravá-las em disco ou enviá-las pela rede (como ocorre para o algoritmo LR). No caso do algoritmo RR, ele também armazena triplas recebidas pela rede.

Ordenação pelo *Quicksort*

Caso seja especificado que o arranjo deve ser ordenado utilizando-se o *Quicksort*, sua estrutura interna será composta basicamente por um único vetor A, que conterá as triplas $\langle t, f, d \rangle$. Isso possibilita a utilização da função `qsort()`, que deve receber como parâmetro um ponteiro para um vetor de elementos em posições contíguas de memória. Entretanto, devido a restrições de alinhamento de memória, essa implementação faz com que cada tripla consuma 12 bytes, apesar dos componentes t, f e d ocuparem, respectivamente, 4, 2 e 4 bytes, totalizando 10 bytes.

Quando uma tripla é inserida no arranjo, seus componentes são copiados para a primeira posição disponível no vetor A e o apontador para a essa posição é incrementado. Se esse apontador exceder o tamanho do vetor, ele é ordenado invocando-se a função `qsort()`. O critério de ordenação, apresentado na Seção 2.1.4, página 21, é ordenar primeiramente segundo os elementos t, em ordem crescente. Triplas com o mesmo componente t são ordenadas por $f_{t,d}$, em ordem decrescente. Quando também os $f_{t,d}$ são iguais, ordenam-se as triplas pelo componente d, em ordem crescente. Os elementos, já ordenados, são então percorridos seqüencialmente a partir do início do vetor e despachados para o objeto **Proxy**. No caso da execução seqüencial, a tripla é inserida em um objeto **CodeBlock**, que a codificará e a armazenará em um *run* no disco.

A alternativa de ordenação pelo *Quicksort*, embora simples e intuitiva, mostra-se ineficiente quando se têm um número muito grande de elementos no arranjo de triplas. O fato é que os

componentes t (termo) e f (freqüência), pelos quais as triplas devem ser ordenadas, possuem uma distribuição muito tendenciosa. Há muitas triplas relativas a um número pequeno de termos e poucas triplas referentes aos termos restantes. Da mesma forma, para a lista invertida de um dado termo, há muitas triplas com freqüências baixas e poucas triplas com altas freqüências. Esse comportamento, modelado pela Lei de Zipf [90], faz com que o comportamento do *Quicksort* seja insatisfatório, chegando próximo ao seu pior caso de $O(n^2)$, como mostrado pela Figura 5.14(c), na Seção 5.3.2. A solução é explorar essa distribuição tendenciosa para se buscar uma ordenação eficiente, cujo comportamento se aproxime, se possível, do linear.

Ordenação linear

De forma a se buscar uma ordenação mais eficiente das triplas, foi criado um método de ordenação que procura explorar a distribuição de termos e freqüências em arquivos invertidos e exibir um comportamento linear sobre o número de elementos no arranjo. Caso, durante a criação do arranjo de triplas, seja especificado que a ordenação deve ser feita através desse método, são alocados três vetores internos para o armazenamento: *Freq*, para as freqüências, *Doc*, para os documentos, e *Next*, para os ponteiros para o próximo elemento da lista invertida de um termo. Com isso, não há mais a ineficiência de armazenamento devido ao alinhamento de memória.

Entretanto, a implementação dessa ordenação linear requer que outros quatro vetores auxiliares sejam alocados: *First*, *Last*, *Size* e *MaxFreq*. Os três primeiros armazenam inteiros de 4 bytes, enquanto que o último contém freqüências de 2 bytes. Esses vetores são utilizados para a implementação de listas encadeadas para os termos da coleção e contém cada um v elementos, onde v é o tamanho do vocabulário.

Por ocuparem uma quantidade de memória proporcional ao número de termos no vocabulário, os vetores auxiliares não representam um gasto em memória que constitua uma desvantagem para a ordenação linear. Pelo contrário, levando-se em conta a economia obtida armazenando-se cada tripla em 10 bytes em vez de 12, a ordenação linear pode consumir menos memória que a ordenação pelo *Quicksort*. Por exemplo, em uma estação de trabalho com 128 Mbytes de memória, na qual a capacidade do arranjo seja estabelecida em 10.000.000 elementos, a ordenação pelo *Quicksort* consumiria $12 \times 10.000.000 = 120.000.000$ bytes de memória. Supondo-se um vocabulário com 1.000.000 de termos (típico de uma coleção de alguns gigabytes), a ordenação linear consumiria $10 \times 10.000.000 + (4+4+4+2) \times 1.000.000 = 114.000.000$ bytes.

O princípio-chave da ordenação linear é criar, sobre o próprio arranjo de triplas, listas invertidas distintas para cada termo da coleção. Isso dispensa o componente t, que identifica a qual termo pertence um elemento, mas demanda um ponteiro entre as tuplas, representado pelo vetor *Next*. Assim, quando o arranjo se tornar cheio e for ordenado, como descrito a seguir, ele pode ser percorrido lista a lista, a partir do termo 0 até o de índice $v - 1$, seguindo-se os ponteiros

Vocabulário	*Size*	*First*	*Last*	*MaxFreq*
a	4	0	16	2
ainda	3	9	13	1
anda	3	2	4	1
aonde	3	3	15	1
onda	5	1	17	2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Next	5	6	4	14	13	7	8	16	11	10	12	17			15			
Freq	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	2	2
Doc	1	1	1	2	2	3	3	4	4	4	5	5	6	6	7	8	9	9

Figura 4.5: Configuração do arranjo após a inserção das triplas da coleção exemplo.

Next. Fazem-se necessários também os ponteiros *First* e *Last*, respectivamente, para o primeiro e o último elemento de cada lista, assim como o vetor *Size*, que indica o tamanho de cada lista. Já *MaxFreq* é um vetor auxiliar usado para determinar a maior freqüência relativa $f_{t,d}$ de cada termo t em um documento d da coleção.

Quando uma nova tripla $\langle t, f, d \rangle$ é inserida no arranjo, identifica-se o índice da primeira posição livre, i. Os elementos f e d são então acomodados, respectivamente, nas posições *Freq*[i] e *Doc*[i], enquanto *Next*[i] recebe um valor inválido, indicando que o elemento inserido é o último da lista. Se a lista do termo t não possui elementos, atribui-se a *First*[t] o índice i. Caso contrário, *Next*[*Last*[t]] é quem passa a apontar para esse índice. Atualiza-se então *Last*[t] com o mesmo i e incrementa-se *Size*[t]. Se a freqüência f for maior que a maior freqüência relativa já encontrada para t, então *MaxFreq*[t] é atualizado com f. A Figura 4.5 mostra a configuração de um arranjo após a inserção de todas as triplas da coleção exemplo, cujo arquivo invertido é exibido na Figura 2.2.

Se o índice i excede o tamanho alocado para os vetores *Freq*, *Doc* e *Next*, os elementos devem ser ordenados e despachados. Nesse ponto, o arranjo contém um conjunto de **listas invertidas parciais**, isto é, listas de ocorrências relativas apenas ao trecho do texto que produziu as triplas em memória. De forma semelhante ao critério definido na página 21, essas listas devem ser ordenadas de forma descrescente pelas freqüências f e, para as triplas com freqüências iguais, a ordenação deve ser crescente, pelos documentos d.

A ordenação é implementada da forma descrita a seguir. Para cada termo do vocabulário, verifica-se o tamanho de sua lista no arranjo. Se ela é vazia (o termo não ocorreu naquele trecho do texto) ou possui apenas um elemento, já está ordenada. Esse corte permite que várias triplas não sejam sequer inspecionadas durante a ordenação. Se a lista parcial do termo possui apenas

dois elementos, basta compará-los e verificar se estão na ordem correta. Caso negativo, suas posições são trocadas, manipulando-se os ponteiros *First*, *Last* e *Next*. Considerando-se em separado os casos acima, explora-se o fato da maioria das listas invertidas, sobretudo as listas parciais presentes no arranjo, possuírem poucos elementos.

Para as listas com três ou mais elementos, a ordenação procura explorar outra característica de arquivos invertidos: a maioria das freqüências relativas $f_{t,d}$ são baixas e apresentam uma distribuição bastante tendenciosa, valendo, em sua maior parte, 1. Emprega-se então uma ordenação por distribuição, utilizando espaço adicional proporcional ao tamanho da lista. Define-se uma freqüência limite f_{lim} (que no sistema **DIG** corresponde a 4) e, sobre a lista invertida L_t sendo ordenada, criam-se $f_{\text{lim}} + 1$ sublistas $\ell_1, \ell_2, \ldots, \ell_{f_{\text{lim}}+1}$.

Percorre-se a lista L_t e, para $1 \leq \varphi \leq f_{\text{lim}}$, faz-se cada sublista ℓ_φ apontar para os elementos $\langle f_{t,d}, d \rangle$ de L_t tais que $f_{t,d} = \varphi$, enquanto que a sublista $\ell_{f_{\text{lim}}+1}$ passa a indicar os elementos para os quais $f_{t,d} > f_{\text{lim}}$. Os ponteiros *Next* são então rearranjados de forma que os primeiros elementos de L_t sejam aqueles listados em $\ell_{f_{\text{lim}}+1}$, seguidos daqueles em $\ell_{f_{\text{lim}}}$ e assim até os elementos em ℓ_1. A lista L_t estará, nesse ponto, quase ordenada, restando-se apenas rearranjar seus primeiros elementos, apontados por $\ell_{f_{\text{lim}}+1}$.

Esses elementos são ordenados através de uma técnica linear estável, que também requer $O(n)$ em espaço adicional: o *list radix sort*, descrito em [42, Seção 5.2.5]. Esse método funciona inspecionando os bits dos elementos a serem ordenados, do menos significativo para o mais significativo. Dados dois limites b_{min} e b_{max}, para cada bit b tal que $b_{\text{min}} \leq b \leq b_{\text{max}}$, todos os componentes f dos elementos da lista são inspecionados. Se o valor de b para um dado elemento é 0, copia-se o seu índice uma pilha P_0, caso contrário, para uma pilha P_1.

Após inspecionados todos os elementos para o bit b, os ponteiros *Next* da lista são rearranjados, de forma que os elementos para os quais b vale 0 sejam copiados para o fim da lista e aqueles cujo bit vale 1 sejam copiados para o início, para que as freqüências sejam dispostas em ordem decrescente. O rearranjo dos ponteiros é feito de forma estável, isto é, se um elemento x precedia um elemento y antes de se inspecionar o bit b e esses elementos são tais que $x[b] = y[b]$, então x continuará a preceder y após a inspeção de b. Isso é feito para que não seja preciso considerar os componentes d na ordenação. Como os documentos são processados sempre em ordem crescente (o primeiro documento a ser processado é o de identificador 0, seguido do documento 1 e assim em diante), as triplas no arranjo já se encontram ordenadas em relação aos componentes d e um método estável de ordenação permite que eles sejam desconsiderados. Percorridos todos os bits entre b_{min} e b_{max}, os elementos da lista estarão em ordem decrescente.

Resta a questão sobre como identificar os limites b_{min} e b_{max}. As triplas cujo componente f é menor ou igual a f_{lim} já se encontram ordenadas, portanto, só é preciso considerar os bits que diferenciam as freqüências superiores a f_{lim}. Logo, toma-se $b_{\text{min}} = \lfloor \log f_{\text{lim}} \rfloor$. No sistema **DIG**, escolhe-se $f_{\text{lim}} = 4$, portanto tem-se $b_{\text{min}} = 2$. De forma análoga, não é preciso considerar bits que somente diferenciariam freqüências superiores às que se têm na lista invertida de um termo.

Nesse ponto utiliza-se a informação no vetor *MaxFreq* e toma-se $b_{\max} = \lceil \log MaxFreq[t] \rceil$. Portanto, a ordenação da sublista $\ell_{f_{\lim}+1}$ acontece em $\lceil \log MaxFreq[t] \rceil - \lfloor \log f_{\lim} \rfloor$ passadas. De acordo com as características de listas invertidas, em geral $MaxFreq[t]$ é um valor pequeno e há poucos elementos na lista $\ell_{f_{\lim}+1}$, logo, essa etapa final da ordenação é também bastante rápida.

4.4.3 Blocos de código

O sistema **DIG** realiza todas as transferências de triplas entre a memória e os dispositivos de entrada/saída (disco e rede) através de blocos, de forma a evitar ineficiências com excessivas chamadas de sistema. A classe *CodeBlock* encapsula essas primitivas de entrada e saída, provendo ainda a possibilidade de compressão dos dados. Na criação de um objeto dessa classe, especificam-se diversas opções de configuração, entre elas, se o bloco será responsável por entrada ou saída, se ele tratará triplas $\langle t, f, d \rangle$ ou duplas $\langle f, d \rangle$, ou se ele comprimirá ou não os dados. Informa-se também número de bytes por bloco e o descritor do arquivo a ser usado para entrada ou saída, que pode corresponder a um arquivo em disco ou um *socket* para comunicação via rede.

Caso o bloco tenha sido criado para a saída de dados, o método *encode*() é utilizado para a codificação de tuplas. Se ele não foi configurado para compressão, as tuplas são armazenadas diretamente em sua área de dados; senão, é feita uma codificação por intervalos, como descrito na Seção 2.1.4, página 22. Quando o bloco torna-se cheio, ele é automaticamente gravado no dispositivo configurado, através do método *dump*(). De forma análoga, para um bloco de entrada de dados, a interface para a decodificação de tuplas é o método *decode*(), que extrai uma tupla do bloco, decodificando-a se necessário. Caso se esgotem os dados disponíveis em um bloco de entrada, novas tuplas são automaticamente carregadas para a memória através do método *load*().

As rotinas de codificação e decodificação devem ser tão eficientes quanto possível, já que cada tripla encontrada no texto é codificada na geração do arquivo invertido temporário, decodificada na intercalação e recodificada na geração do arquivo invertido final. De forma a se obter essa eficiência, utiliza-se uma estratégia comum para a aceleração de operações, a pré-computação de valores. Para a decodificação, entretanto, não é possível realizar nenhum tipo de pré-computação, já que não é possível prever quais códigos serão encontrados na entrada até que os bits tenham sido efetivamente lidos. Por outro lado, bastante computação pode ser economizada calculando de antemão os códigos para os números inteiros mais baixos, já que a maioria dos valores codificados são pequenas diferenças inteiras. Assim, na primeira instanciação de um objeto da classe **CodeBlock**, cria-se um *cache* de códigos comum a todos os objetos dessa classe, o que permite grandes economias de tempo na compressão e recompressão dos arquivos invertidos.

4.4.4 Intercalação das triplas

A intercalação dos *runs* produzidos durante o processamento do texto também é importante para o desempenho da indexação, já que todas as triplas armazenadas no arquivo temporário devem ser lidas, ordenadas e gravadas no arquivo invertido final. Para que essa atividade seja realizada eficientemente, várias técnicas são empregadas, possibilitando a ordenação rápida das triplas e a realização de um número pequeno de *seeks* a disco, entre outras otimizações.

Primeiramente, informações sobre os *runs* são coletadas à medida em que eles são produzidos e armazenadas em um objeto da classe **RunIndex**. Tais informações incluem o número de blocos em cada *run* e o *offset* de início do *run* no arquivo temporário. Esse índice de *runs* é construído no início do processamento do texto, com um tamanho R estimado por $R = x|\langle t, f, d\rangle|/M$, mas pode ser redimensionado quando necessário.

O processo de intercalação começa com a instanciação de um objeto da classe **Merge**. O construtor dessa classe cria o *heap* utilizado na ordenação das triplas e as listas de blocos responsáveis pela leitura dos dados do disco. Os primeiros blocos de cada *run* são carregados para a memória e a primeira tripla de cada um deles é inserida no *heap*. Este é rearranjado de forma que a tripla de maior prioridade, segundo os critérios de ordenação estabelecidos, seja posicionada no topo, permitindo o início da intercalação.

O funcionamento da intercalação é bastante simples. Enquanto o *heap* não se torna vazio, remove-se o elemento no seu topo, isto é, aquele de maior prioridade, que é despachado através do objeto **Proxy**. Procura-se então decodificar a próxima tripla do mesmo *run* do qual o elemento despachado foi retirado. Caso todos os elementos desse *run* já tenham sido decodificados, procura-se por um próximo *run* disponível e decodifica-se sua próxima tripla. Se, por outro lado, não há mais *runs* com elementos disponíveis, toma-se a tripla na última posição do *heap*, que tem seu tamanho decrementado de 1. A tripla obtida é inserida no topo do *heap*, que é rearranjado para se restaurar sua propriedade de ordenação e se continuar a intercalação.

4.5 Algoritmos Distribuídos

Nesta seção descreve-se a implementação das classes e métodos ligados aos algoritmos distribuídos e como essas entidades colaboram para a obtenção tanto do vocabulário como do índice invertido para a coleção global. A Seção 4.5.1 descreve a implementação da fase de levantamento do vocabulário global de uma coleção distribuída, que é comum a todos os algoritmos implementados neste trabalho. A Seção 4.5.2 apresenta os critérios para distribuição (envio e recepção) das triplas que permitem a geração do arquivo invertido global através de cada um dos algoritmos LL, LR e RR.

4.5.1 Obtenção do vocabulário global

O levantamento do vocabulário global, primeira fase da execução dos algoritmos distribuídos implementados, é bastante semelhante à fase correspondente do algoritmo seqüencial, descrita na Seção 4.3. Cada máquina começa processando o texto em disco e obtendo seu vocabulário local. Os vocabulários de cada máquina são então trocados pela rede e intercalados, segundo os passos do Algoritmo 3.1, obtendo-se o vocabulário global na máquina de índice 0. Essa máquina gera a função *hash* perfeita para os termos e a distribui às demais.

Não há classes específicas para a implementação desse processo nem é necessário o emprego de *multithreading* ou programação concorrente. A classe **NormHash**, ao completar o levantamento do vocabulário local em uma máquina π, verifica se ela está participando de uma execução distribuída checando se o número de processadores p é maior que um. Caso afirmativo, ela executa a Subfase 3 do Algoritmo 3.1. Nessa fase, o vocabulário global para p processadores é computado em $\lceil \log p \rceil$ passos, em cada qual a máquina π pode enviar o vocabulário nela armazenado ou receber o vocabulário de outra máquina.

Quando, em um dado passo desse algoritmo, uma máquina π envia seu vocabulário a outra máquina τ, as *strings* armazenadas na classe **Vocab** de π são transferidas para τ através de um *socket* de comunicação. O vocabulário transmitido é então intercalado ao de τ. Essa intercalação é implementada pelo método *merge*() da classe **NormHash**, que percorre os vocabulários V_π e V_τ, ordenados previamente à transmissão, comparando diretamente os termos em cada um deles e copiando-os de forma ordenada para um vocabulário resultante V_{res}. Este último será armazenado na máquina τ, substituindo V_τ.

Ao final desse algoritmo, a máquina 0, como sugerido pela Figura 3.2, conterá o vocabulário global da coleção, ordenado lexicograficamente. A função *hash* perfeita pode então ser gerada, assim como feito para a geração seqüencial. A máquina 0 transmite às demais os dados necessários para o uso dessa função, ou seja, a tabela de mapeamento g e os vetores aleatórios $h_0, \ldots, h_{d-1}$. Todas as máquinas escrevem esses dados em seus discos locais, para permitir a execução da fase de indexação.

Ainda deve-se realizar uma nova troca de informação entre as máquinas para a preparação da fase seguinte. Como o arquivo invertido a ser gerado é global e deve ser capaz de apontar para todos os documentos da coleção distribuída, os identificadores desses documentos devem ser únicos para todas as máquinas da rede. Se cada máquina π numerar seus documentos locais através de inteiros $0, 1, \ldots, c_\pi$, haverá colisões de identificadores quando as triplas forem trocadas pela rede. Para se evitar isso, determina-se a partir de qual número cada máquina irá identificar seus documentos locais. Isso é feito através do método *computeDocBases*() da classe **DigHash**, descrito na página 60. Cada máquina envia o número de documentos de sua coleção local para a máquina 0, que computa o valor inicial para a numeração de documentos em cada outra máquina e o envia pela rede. Recebendo esse valor, as demais máquinas o armazenam em disco e terminam a execução da fase de levantamento do vocabulário.

4.5.2 Geração do arquivo invertido global

Como notado nas Seções 3.3, 3.4 e 3.5, os algoritmos distribuídos propostos neste trabalho diferem, conceitualmente, apenas no momento em que as triplas são trocadas pela rede. O sistema **DIG**, de forma a permitir a implementação eficiente dessas três abordagens, bem como do seqüencial, e o reuso das classes comuns a eles, procura encapsular a inteligência que controla a troca de triplas pela rede na classe **Proxy**, descrita na página 68. A arquitetura do processo de indexação é basicamente a mesma, segundo ilustrado nas Figuras 3.3, 3.5 e 3.7.

Na sua criação, um objeto da classe **Proxy** é configurado de acordo com o tipo do algoritmo sendo executado, seja ele o seqüencial ou uma das versões distribuídas. Nos três momentos ao longo da indexação em que as triplas são identificadas, o método *dispatch*() da classe **Proxy** é invocado e decide o que deve ser feito, segundo as opções abaixo.

Momento 1: As triplas são identificadas pelo tipo abstrato Dicionário, representado pela tabela *hash* perfeita, como citado na Seção 4.4.1, página 85 e ilustrado pela Figura 3.7.

Algoritmo RR: Triplas enviadas pela rede para a máquina de destino.

Demais algoritmos: Triplas inseridas no *buffer* de triplas da máquina local.

Momento 2: As triplas são identificadas pelo *buffer* implementado pela classe **Array**, logo após ele se tornar cheio e ser ordenado. Esse instante é descrito na Seção 4.4.2, página 86 e destacado pela Figura 3.5.

Algoritmo LR: Triplas enviadas pela rede para a máquina de destino.

Demais algoritmos: Triplas inseridas em um bloco que as armazenará no arquivo temporário, em disco local.

Momento 3: Identificam-se as triplas durante a intercalação dos *runs* em disco, assim que elas são removidas do *heap*, como mencionado na Seção 4.4.4, página 91. Esse momento é ilustrado pela Figura 3.3.

Algoritmo LL: Triplas enviadas pela rede para a máquina de destino.

Demais algoritmos: Triplas inseridas em um bloco que as armazenará no arquivo invertido final, no disco local.

Quando é dito que as triplas são enviadas pela rede, na realidade, por questões de eficiência, elas são disponibilizadas em um *buffer* para a *thread* de envio, descrita logo a seguir. De forma análoga, há três momentos correspondentes nos quais triplas são recebidas pela rede. Isso é feito de forma assíncrona pelas *threads* de recepção, que incorporam as triplas ao fluxo de dados em cada um desses pontos da execução como se elas tivessem sido identificadas no texto local. A organização das *threads* de recepção e a forma pela qual elas contribuem para a eficiência e transparência do processo são descritas ainda nesta seção.

Envio de triplas

A necessidade do envio de triplas a outra máquina pode ser identificada em três diferentes momentos da execução dos algoritmos distribuídos. Se, sempre que essa transmissão deve ocorrer, o processo de indexação se bloquear, esperando o fim da transferência, seu desempenho será comprometido, pois a rede pode ser lenta ou encontrar-se sobrecarregada. A solução é o uso de uma *thread* específica para o envio de triplas pela rede, a *sendThread* apresentada na descrição da classe **Proxy**, na Seção 4.1, página 68. Essa *thread* somente ocupará a CPU quando houver triplas para serem enviadas e pode ficar bloqueada esperando pela transmissão, sem comprometer a execução do processo de indexação.

A *thread* principal, responsável pela indexação, deve se comunicar com a *thread* de envio através de alguma área compartilhada de memória, onde as triplas a serem transmitidas são disponibilizadas. Essa é a função da classe **ProxyBuffer**, descrita na página 69: proporcionar um *buffer* compartilhado, no qual a *thread* de indexação possa inserir as triplas a serem transmitidas sem a necessidade de se bloquear esperando a resposta da rede. Da mesma forma, a *thread* de envio pode remover as triplas desse *buffer* e transmiti-las pela rede, dispensando qualquer outra sincronização com a *thread* principal.

Um objeto *PB* da classe **ProxyBuffer** é implementado internamente como uma lista circular de *PB.max* elementos, na qual o primeiro elemento é apontado por *PB.first* e o último, por *PB.last*. As funções auxiliares *full*() e *empty*() utilizam esses apontadores para testar se o *buffer* encontra-se cheio ou vazio, respectivamente. O acesso ao objeto é proporcionado pelos métodos *insert*(), executado pela *thread* principal e detalhado pelo Algoritmo 4.2, e *remove*(), invocado sempre pela *thread* de envio, descrito pelo Algoritmo 4.3. Ambos os métodos devem, em primeiro lugar, requisitar acesso exclusivo ao *buffer* através da primitiva de sincronização *lock*(). Outras duas primitivas de programação concorrente, *wait*() e *signal*(), são utilizadas para lidar com as condições *PB.notFull* e *PB.notEmpty*, que os métodos de inserção e remoção devem observar antes de prosseguir.

A *thread* de envio, por sua vez, executa um laço no qual ela sempre tenta remover uma tripla do *buffer* e codificá-la em um bloco de dados. Este último somente será enviado para a máquina de destino quando se tornar cheio, funcionando, portanto, como um outro *buffer* e evitando a ineficiência de se enviar uma quantidade grande de pequenos itens pela rede. O uso de compressão nesse bloco permite também a redução do volume de dados transferidos, o que não seria possível caso as triplas fossem enviadas uma a uma. Deve-se observar, entretanto, que o uso de compressão somente é possível para os algoritmos LL e LR, que ordenam suas triplas antes da transmissão. O algoritmo RR, que envia as triplas à medida em que elas são identificadas no texto local, sem previamente ordená-las, não permite compressão nos blocos de tráfego de dados pela rede. Isso implica em uma desvantagem para esse algoritmo, como mostrado nos resultados da Seção 5.4.2.

Outra particularidade nesse ponto é devida ao algoritmo LR. De acordo com a Seção 3.4,

logo após ordenar seu arranjo local, uma máquina executando esse algoritmo envia as triplas para as demais $p-1$ máquinas como $p-1$ "sub-*runs*", que são armazenados diretamente no disco para serem posteriormente intercalados. A *thread* de envio precisa então identificar a qual sub-*run* pertence um determinado bloco de código, para que a máquina receptora não o armazene juntamente com os próximos dados a serem transmitidos. Isso é feito enviando, antes de cada bloco, um inteiro indicando a qual sub-*run* pertence aquele bloco. Quando esse inteiro muda, a máquina receptora pode identificar que um novo *run* está sendo transmitido.

```
Entradas: ProxyBuffer PB, a tripla ⟨t, f, d⟩
  lock(PB)
  enquanto PB.full() faça
    wait(PB.notFull)
  fim enquanto
  PB[PB.last] ← ⟨t, f, d⟩
  se PB.empty() então
    signal(PB.notEmpty)
  fim se
  PB.last ← (PB.last + 1) mod PB.max
  unlock(PB)
```

Algoritmo 4.2: Inserção de uma tripla no *buffer* de envio.

```
Entradas: ProxyBuffer PB
Saídas: A tripla ⟨t, f, d⟩
  lock(PB)
  enquanto PB.empty() faça
    wait(PB.notEmpty)
  fim enquanto
  ⟨t, f, d⟩ ← PB[PB.first]
  se PB.full() então
    signal(PB.notFull)
  fim se
  PB.first ← (PB.first + 1) mod PB.max
  unlock(PB)
```

Algoritmo 4.3: Remoção de uma tripla do *buffer* de envio.

Recepção de triplas

Ao passo em que os algoritmos distribuídos pouco diferem na forma pela qual enviam-se as triplas, eles apresentam diferenças significativas na recepção das mesmas. Isso porque, quando uma máquina recebe triplas pela rede, ela deve incorporá-las ao fluxo de dados da indexação

como se elas fossem identificadas no texto local. Como, para cada algoritmo, essa incorporação se dá em um ponto diferente da execução, daí as diferenças entre as rotinas de recepção.

Ainda assim, a organização das *threads* responsáveis pela recepção das triplas é a mesma para todos os algoritmos distribuídos. Em uma indexação com p máquinas, cada uma cria $p-1$ *recvThreads*, apresentadas na descrição da classe **Proxy**. Cada uma dessas *threads* procura carregar um bloco de código com dados recebidos pela rede de uma das $p-1$ máquinas colaborando no processo de indexação. Quando se completa a leitura do bloco, as triplas nele contidas podem incorporar o fluxo de dados local do algoritmo, da forma descrita a seguir, enquanto que a *thread* volta a esperar por mais dados.

Algoritmo LL. As triplas são intercambiadas no momento 3, segundo a categorização apresentada anteriormente nesta mesma seção. Não é necessário decodificar as triplas presentes nos blocos lidos da rede; eles são diretamente armazenados em disco, formando um segundo arquivo temporário, denominado F'' pelo Algoritmo 3.2. Na realidade, F'' é implementado como um conjunto de p arquivos, cada um contendo os dados produzidos em uma das máquinas participando da indexação, incluindo a máquina local. Assim, as *threads* de recepção não precisam competir pelo acesso de escrita a um único arquivo e é mais simples armazenar as informações sobre os dados a serem intercalados, pois cada arquivo conterá um único *run*.

Algoritmo LR. A comunicação via rede ocorre quando se descarrega o *buffer* de triplas, no momento 2, como descrito previamente. Os blocos são recebidos pelas *recvThreads*, sendo sempre precedidos por um inteiro identificando o *run*. Quando o identificador do *run* muda, sabe-se que o bloco faz parte de um novo *run* e uma nova entrada no objeto **RunIndex** é criada. Assim como feito no algoritmo LL, os blocos vindos de cada uma das p máquinas, contando com a máquina local, são armazenados em p arquivos distintos, evitando as complicações devidas ao acesso concorrente. Como cada máquina produz R *runs*, cada arquivo desses armazenará R sub-*runs* ordenados e o algoritmo LR deverá executar uma intercalação por $p \times R$ caminhos, como já previsto.

Algoritmo RR. Esse é o algoritmo mais complexo em relação à recepção de triplas, pois não há como evitar a competição por recursos entre as *threads*. As triplas são trocadas pela rede ao longo de todo o processamento do texto e, ao serem recebidas, devem ser inseridas no arranjo, juntamente com os dados produzidos localmente. Dessa forma, as *recvThreads* devem concorrer com a *thread* principal pela inserção no arranjo, que é um recurso compartilhado por todas elas. Esse fato foi destacado na Seção 3.5 e já havia sido previsto em [62]. A seguir discutem-se as abordagens para lidar com essa concorrência.

Uma primeira abordagem para o controle da concorrência ao arranjo de triplas é utilizar *locks* exclusivos a cada inserção. Sempre que uma *thread* de recepção ou a *thread* principal pro-

cura inserir uma tripla no arranjo, ela deve anteriormente invocar a primitiva de sincronização *lock*() para obter acesso exclusivo ao método de inserção, considerado uma seção crítica. Quando esse acesso é obtido, ela pode modificar o estado interno do arranjo e inserir o novo elemento, sem se preocupar com perturbações provocadas por outras *threads*.

Essa abordagem, embora bastante intuitiva, é muito ingênua e leva a uma grande ineficiência na execução do algoritmo RR, pois as *threads* passam muito tempo competindo pelo acesso ao arranjo. Experimentos realizados em estágios precoces do desenvolvimento mostram que, utilizando essa técnica, o algoritmo RR pode ser até 3 vezes mais lento que o algoritmo LR. Notou-se também que, dentre todos os ciclos de CPU gastos no algoritmo RR com essa abordagem, menos de 30% eram gastos executando código do programa. Os ciclos restantes eram gastos no *kernel* dos sistema operacional, certamente nas primitivas de sincronização, esperando pelo acesso exclusivo à seção crítica.

Uma solução para a excessiva contenção ocasionada pela abordagem anterior é análoga ao que foi feito para os algoritmos LL e LR, quando se criou um arquivo temporário distinto para cada *thread*: definir uma área no arranjo de triplas reservada para a inserção por cada *thread*. Como o número de *threads* realizando inserções é p (pois há uma *thread* principal e $p-1$ *recvThreads*, o arranjo é dividido em p pedaços. As inserções não necessitam mais acesso exclusivo e podem ocorrer simultaneamente. Quando um dos pedaços torna-se cheio, há duas alternativas: ordenar o arranjo como um todo, mesmo que os demais pedaços ainda possuam posições vazias, ou esperar para que todos os pedaços tornem-se cheios antes de se proceder a ordenação.

A primeira dessas alternativas requer que, quando a ordenação for disparada, não haja nenhuma *thread* inserindo no arranjo, para não se correr o risco deste último encontrar-se em um estado inconsistente. Para isso, a rotina de inserção realiza um travamento compartilhado no arranjo, que permite mais de uma inserção simultâneas, enquanto que o método de ordenação efetua um travamento exclusivo. Esses dois tipos de travamentos são conhecidos na nomenclatura de bancos de dados e sistemas operacionais por *shared locks* e *exclusive locks*, sendo utilizados, respectivamente, para leituras e escritas em registros. Quando uma *thread* enche seu pedaço no arranjo, ela deve esperar que todas as outras *threads* realizando inserções completem essa operação antes de disparar a ordenação. Do momento em que o acesso exclusivo é obtido até o fim da ordenação, as *threads* que tentam efetuar inserções ficam bloqueadas.

Com essa alternativa, o desempenho do algoritmo RR melhorou significativamente, pois ela reduz a seção crítica executada pelas inserções ao incremento de um inteiro, o contador do número de *threads* operando sobre o arranjo. Ainda assim, o simples fato de se invocar as primitivas de sincronização para cada tripla inserida no arranjo representa uma sobrecarga de processamento significativa. A segunda alternativa listada acima, esperar que todos os pedaços do arranjo tornem-se cheios antes de ordená-lo, dispensa a necessidade de se obter qualquer tipo de travamento sobre o arranjo durante as inserções.

Por essa segunda alternativa, todas as *threads* inserem livremente em seus pedaços do arranjo. A primeira *thread* a preencher seu pedaço espera que todas a outras também o façam para proceder a ordenação. À medida que as demais *threads* vão preenchendo seus pedaços, elas passam a esperar que a primeira *thread* ordene o arranjo como um todo. Quando isso ocorre, todas as *threads* são liberadas e voltam a inserir triplas no arranjo. Com isso, não há contenção na inserção no arranjo, pois as *threads* só ficam bloqueadas quando enchem seus respectivos pedaços no arranjo, e não é necessário invocar primitivas de sincronização a cada tripla processada. Por outro lado, as *threads* esperam um tempo maior pela ordenação do arranjo, pois todas elas devem preencher seus pedaços antes que as triplas sejam ordenadas.

Essa alternativa não proporcionou ganho significativo de desempenho em relação à anterior e ainda sofre de um problema mais grave: ela é propensa à ocorrência de *deadlocks*. Um cenário que mostra essa propensão é descrito a seguir. Supondo duas máquinas P_0 e P_1 participando da execução do algoritmo RR, se a *thread* principal de P_0 é a primeira a preencher seu pedaço no arranjo, ela passa a esperar que a *thread* responsável pela recepção das triplas de P_1 também o faça (ou termine sua execução) para proceder a ordenação. Entretanto, se, na máquina P_1, a *thread* principal também foi a primeira a preencher sua parte do arranjo, ela também passa a esperar que a outra *thread*, que recebe dados de P_0, preencha seu pedaço ou termine. Mas P_0 não enviará nenhum dado a P_1, pois sua *thread* principal, que identifica as triplas no arranjo local, está esperando dados de P_1. Nesse caso, a máquina P_0 está esperando dados de P_1 e vice-versa, logo, o sistema encontra-se em *deadlock*. Sendo assim, o sistema **DIG** emprega o sistema de *locks* compartilhados e exclusivos para controlar o acesso ao arranjo de triplas.

Outras alternativas para melhorar o desempenho da inserção concorrente de triplas no arranjo foram projetadas e algumas chegaram a ser testadas. Por exemplo, procurou-se reduzir o número de *threads* de recepção a apenas uma, de forma a reduzir a competição pelo arranjo. Entretanto, a complexidade dessa *recvThread* única é bem maior, pois ela deve realizar a leitura simultânea dos $p-1$ *sockets* de recepção. Isso foi implementado utilizando-se multiplexação da entrada por esses *sockets*. Assim, a *thread* de recepção deveria escutar em um *socket*, lendo os dados por ele enviados, ou até obter um *timeout*, então deveria passar a escutar em outro *socket*. Essa alternativa não resolveu o problema de acesso ao arranjo de triplas (pois continua havendo a necessidade de sincronização entre a *thread* principal e a de recepção) e ainda acrescentou um processamento extra ao processo de recepção de dados, tornando-o semelhante a uma espera ocupada.

Uma outra possibilidade de implementação cogitada foi o uso de interrupções assíncronas para, quando uma *thread* preenche seu pedaço no arranjo, notificar as demais de forma que elas não realizem inserções. Essa alternativa liberaria as *threads* de invocarem as primitivas de sincronização a cada inserção no arranjo. Por outro lado, como uma *thread* não poderia ser interrompida durante uma inserção (pois, nesse caso, o arranjo seria deixado em uma condição inconsistente), todas as *threads* deveriam modificar o *handler* da interrupção a cada inserção no

arranjo. Portanto, essa alternativa não elimina o custo de uma chamada de sistema adicional a cada inserção de tripla e foi descartada.

A ordenação do arranjo é outra questão que deve ser tratada de forma especial no algoritmo RR. Caso a ordenação seja feita pelo *Quicksort*, as triplas devem ser movidas para o início do arranjo, já que pode haver pedaços não totalmente preenchidos. Para a ordenação linear, cada termo deve manter p listas distintas, uma para cada pedaço, que devem ser concatenadas antes do início da ordenação. Isso implica em uma maior demanda de espaço: para um arranjo com 10.000.000 elementos, um vocabulário de 1.000.000 termos e 4 máquinas, as estruturas para a ordenação linear ocupariam 156.000.000 bytes, 42.000.000 bytes a mais que o espaço gasto pelos demais algoritmos.

Capítulo 5

Resultados Analíticos e Experimentais

Neste capítulo apresentam-se os resultados obtidos através da análise e experimentação do sistema **DIG**. A Seção 5.1 caracteriza o ambiente de experimentação, levantando os parâmetros da coleção e do ambiente listados nas Tabelas 1.1 e 1.2. A Seção 5.2 descreve brevemente a invocação do sistema **DIG**, como foram executados os experimentos e coletados os dados apresentados. Os resultados da geração seqüencial de arquivos invertidos são exibidos na Seção 5.3, enquanto que a geração distribuída encontra-se analisada na Seção 5.4. Cada uma dessas seções valida, para ambas as fases da indexação, os modelos apresentados nos Capítulos 2 e 3.

5.1 Ambiente de Experimentação

Descreve-se aqui o ambiente utilizado na experimentação do sistema **DIG** e na validação das análises elaboradas ao longo deste trabalho. As máquinas são caracterizadas qualitativamente e então definem-se os parâmetros de entrada dos modelos propostos para os algoritmos desenvolvidos.

A rede empregada nos experimentos é composta por quatro PCs, com configurações típicas de estações de trabalho atuais. As máquinas possuem configurações de hardware e software idênticas:

- Processador Intel Pentium III 500 MHz.
- 128 Mbytes de memória RAM.
- Disco IDE de 8 Gbytes.
- Placa de rede 3Com 3C905B, 10/100 Mbits.
- Sistema operacional Linux, *kernel* 2.2.13.
- Compilador GCC 2.95.2, com otimizações ativas e biblioteca GLIBC 2.1.2.

As máquinas encontram-se conectadas em uma VLAN através de um *switch* BayStack 450 10/100 Mbits da Bay Networks, o que lhes permite total isolamento de interferências externas e liberdade para comunicação entre qualquer par de máquinas, sem contenção.

Esse ambiente foi testado através de alguns *benchmarks* criados para esse propósito e do próprio sistema **DIG**, que imprime na saída as estatísticas coletadas ao longo de sua execução, como mostra a Figura 5.1. Os parâmetros dos modelos elaborados, introduzidos nas Tabelas 1.1 e 1.2, foram calculados com base nessas estatísticas e seus valores encontram-se listados nas Tabelas 5.1 e 5.2. Por se tratar de um modelo aproximado, que se propõe a formalizar apenas as principais etapas da execução dos programas, os valores levantados são também aproximados, possuindo apenas um ou dois dígitos significativos. Em relação a esses valores, chama atenção o fato do tempo de acesso ao disco ser cerca de 30 vezes menor que o de acesso à rede, o que pode ser atribuído à influência dos *caches* de disco em Linux e à inacurácia na medida do tempo de resposta da rede.

Alguns parâmetros, como n, o número de bytes do texto, M, a memória disponível para indexação e p, o número de processadores executando o algoritmo, são especificados diretamente, via linha de comando, para cada execução do programa e não se encontram listados nessas tabelas. Para os testes, utilizou-se n variando de 100 Mbytes a 3000 Mbytes, com intervalos de 100 Mbytes. O valor de M varia entre 8 Mbytes e 80 Mbytes, com intervalos de 4 Mbytes. A razão para não se empregar uma parte maior dos 128 Mbytes de memória física disponível nas máquinas é o consumo devido ao próprio *kernel* do sistema operacional, aos *buffers* de disco, a outros processos e às demais estruturas de dados do próprio sistema **DIG**. O número de processadores p varia entre 1 e 4 nos testes. Outros valores, como o tamanho em bytes dos índices invertidos, f, f' e f'', são derivados das densidades de bits por ponteiro mostradas na Tabela 5.1. Deve-se observar que o tamanho médio das palavras no vocabulário apontado nessa mesma tabela inclui o byte indicador de tamanho da *string*.

5.2 Invocação do sistema DIG

A execução do sistema **DIG** pode ser controlada por um número de opções, processadas pela classe **Option**, descrita na página 61. A seguir são listados os parâmetros aceitos por cada uma das fases da execução, seja ela seqüencial ou distribuída.

- Levantamento do vocabulário: `dighash -b` *baseName* `[-d` *dim*`] [-h] [-m` $m_0, \ldots, m_{p-1}$`] [-s` *seed*`]`
  - `b` *baseName*: especifica a *string* a ser usada como base para todos os nomes de arquivos a serem gerados na execução.
  - `d` *dim*: número de dimensões do grafo a ser usado na geração da função *hash* perfeita. Valores possíveis: 2 e 3.

Símbolo	Valor	Significado
w	$n/(\|w_C\|+1.5)$	Número de palavras na coleção
x	$w/2$	Número de pontos de indexação na coleção
$\|w_C\|$	4.763	Tamanho médio das palavras da coleção
v	Kn^{β}	Número de termos no vocabulário
$\|w_V\|$	9.287	Tamanho médio das palavras do vocabulário
σ	256	Número de caracteres no alfabeto
K	4.8	Parâmetro da Lei de Heaps [37]
β	0.56	Parâmetro da Lei de Heaps [37]
θ	1.85	Parâmetro da Lei de Zipf [90]
$8f/x$	10.492	Densidade de bits no índice final
$8f'/x$	11.829	Densidade de bits no índice temporário
$8f''/x$	11.000	Densidade de bits no índice temporário (algoritmo LL)

Tabela 5.1: Variáveis relativas à coleção local a cada processador.

Símbolo	Valor	Significado
b	4096 bytes	Tamanho do bloco de dados
R	$R = \frac{x\|\langle t,f,d\rangle\|}{M}$	Número de *runs* produzidos na indexação
λ	150	Constante de proporcionalidade do algoritmo linear de ordenação
α^{-1}	2.09	Razão entre vértices e arestas do grafo da OPMPHF
t_r	1.5×10^{-8}s	Tempo médio de transferência do disco por byte
t_s	1.2×10^{-2}s	Tempo médio de *seek* do disco
t_z	2.5×10^{-7}s	Tempo médio de compressão por byte produzido
t'_z	6.0×10^{-7}s	Tempo médio de descompressão por byte fornecido
t_p	1.5×10^{-7}s	Tempo do primeiro *parsing* do texto por byte
t'_p	3.0×10^{-7}s	Tempo do segundo *parsing* do texto por byte
t_n	5.0×10^{-7}s	Tempo de transferência pela rede por byte
t_{ct}	5.0×10^{-9}s	Tempo para comparar duas triplas $\langle t,f,d\rangle$
t_{cs}	3.0×10^{-9}s	Tempo para comparar duas *strings* por byte

Tabela 5.2: Parâmetros associados ao ambiente de operação dos algoritmos.

  - `h`: imprime as opções de uso do programa.
  - `m` $m_0, \ldots, m_{p-1}$: estabelece as máquinas a serem utilizadas em uma execução distribuída. O nome da máquina local deve estar presente.
  - `s` *seed*: especifica uma semente para inicializar o gerador de números aleatórios. Deve ser um valor não-negativo.

- Geração do índice: `digindex -b` *baseName* `[-a` *algoName*`] [-h] [-l] [-t` *memSize*`] [-z]`
  - `b` *baseName*: *string*-base a ser usada para se montar os nomes dos arquivos gerados pela fase anterior e criados pela fase atual.
  - `a` *algoName*: algoritmo distribuído a ser utilizado. Essa opção deve ser especificada se e somente se mais de uma máquina foi incluída pela opção `-m` na fase anterior.
  - `h`: imprime a mensagem de ajuda com as opções de uso.
  - `l`: especifica que deve-se usar a ordenação linear no arranjo de triplas.
  - `t` *memSize*: estabelece o número de bytes de memória a ser destinado ao arranjo de triplas.
  - `z`: indica que deve-se utilizar compressão nas triplas.

Para os experimentos realizados, as chamadas ao sistema **DIG** foram incluídas em *shell scripts* e o tempo de execução foi medido através do comando UNIX `time`, que exibe o tempo de relógio tomado pelo processo, juntamente com o tempo gasto executando código do usuário e do *kernel* e a percentagem de CPU utilizada pelo processo. Exceto onde indicado, o tempo mostrado em todos os resultados desta seção correspondem ao tempo total de relógio gasto pelo programa. No caso da medição do tempo gasto em trechos do programa, foram empregadas macros que invocam a função C `gettimeofday`, permitindo a tomada de tempo com resolução de microssegundos. Um trecho da saída produzida nos experimentos foi incluída na Figura 5.1.

Os dados presentes na saída da execução foram coletados através de um conjunto de *scripts* Perl, que também eram responsáveis por calcular o tempo médio a partir de várias execuções dos programas. Os gráficos foram gerados pelo programa Gnuplot. As entradas para o programa consistiam em arquivos da coleção TREC-3, de todos seus três discos.

5.3 Geração Seqüencial

Nesta seção apresentam-se os resultados obtidos para a geração seqüencial de arquivos invertidos. Além dos gráficos que exibem o tempo de execução do programa e validam o modelo matemático proposto, incluem-se figuras que mostram outras da execução seqüencial e da distribuída, como o crescimento do vocabulário com o texto, os ganhos em tempo de execução obtidos com o uso de compressão ou ordenação linear, entre outras.

```
        *** Statistics from dighash (sequential execution) ***
Size of local text ____________________ 1048644399 bytes
Time to read local text _______________ 12.262s (1.169294e-08 secs/byte)
Time to parse local words _____________ 150.441s (1.434620e-07 secs/byte)
Number of documents in local text _____ 770978
Number of words in local text _________ 160596289
Mean local word length ________________ 4.852 chars/word
Time to sort local vocabulary _________ 6.702s
Time to merge vocabularies ____________ 0.000s
Number of distinct global words _______ 482925
Mean distinct global word length ______ 9.146 chars/word
Space used by global vocabulary _______ 4416596 bytes
Number of graphs generated ____________ 2
Time to generate OPMPHF _______________ 2.452s (5.078167e-06 secs/term)
Number of dimensions of the graph _____ 2
Inverse load factor of g function _____ 2.09
Total memory used by graph ____________ 15627476s
173.24 159.58 11.57 98%

        *** Statistics from digindex - sequential ***
Size of local text ____________________ 1048644399 bytes
Time to read local text _______________ 18.687s (1.782048e-08 secs/byte)
Time to parse local words _____________ 393.772s (3.755055e-07 secs/byte)
Size of buffer for storing triples ____ 67108864 bytes
Number of triples per buffer __________ 6710886
Number of runs ________________________ 12
Time to merge triples _________________ 115.565s
Time to sort triples __________________ 56.183s (7.465984e-07 secs/triple)
Index points in local text ____________ 75251859
Total bytes coded in temp file ________ 114708480 (12.199 bits/point)
Total bytes coded in final file _______ 102871040 (10.940 bits/point)
536.67 500.26 16.92 96%
```

Figura 5.1: Saída dos programas componentes do sistema **DIG**.

5.3.1 Obtenção do vocabulário

Os gráficos que validam a expressão para a duração da fase de levantamento do vocabulário do algoritmo seqüencial são exibidos a seguir, juntamente outras figuras indicando propriedades do vocabulário da coleção e do seu processo de obtenção.

Validação do modelo

Inclui-se, na Figura 5.2, o gráfico do tempo de execução da fase de levantamento do vocabulário contra os valores estimados pelo modelo matemático proposto. Observa-se um ajuste satisfatório proporcionado pelo modelo, confirmando que o tempo de execução desta fase é linear com o tamanho n do texto. Realmente, o termo t_{SEQ_A} da Equação 2.1, que corresponde à obtenção do vocabulário, é $O(n)$, pois, dado que $\beta < 1$, tem-se

$$\begin{aligned} t_{SEQ_{A1}} &= n(t_r + t_p) = O(n) \\ t_{SEQ_{A2}} &= (1.2v \log v)|w_V|t_{cs} = O(n^\beta \log n^\beta) = o(n) \\ t_{SEQ_{A3}} &= O(v) = O(n^\beta) = o(n). \end{aligned}$$

Cumpre notar que as maiores perturbações sofridas pelo tempo de execução para valores de n acima de 2 Gbytes são devidas ao algoritmo aleatório para o cálculo da função *hash* perfeita. À medida que o conjunto de chaves – no caso, o vocabulário da coleção – cresce, as oscilações no número de grafos que devem ser gerados para se obter um grafo acíclico também aumentam, provocando maiores variações no tempo de execução.

Outros gráficos

Apresentam-se, a seguir, gráficos ilustrando outras propriedades da fase de levantamento seqüencial do vocabulário. A Figura 5.3 mostra o tempo de execução dessa fase quando se utiliza um grafo de 2 ou 3 dimensões para a geração da função *hash* perfeita. Nota-se que o tempo de execução para os dois casos é bastante semelhante para a maioria dos tamanhos da entrada, mas que, para os valores acima de 2 Gbytes, a geração por trigrafos mostra-se mais estável. Isso decorre do fato da probabilidade de surgirem ciclos em grafos aleatórios de maiores dimensões é menor quanto maior a dimensão do grafo. É por isso que, para trigrafos, pode-se utilizar um fator α^{-1} de apenas 1.23, ao passo que digrafos requerem um fator de, no mínimo, 2.09.

A Figura 5.4 mostra o crescimento do vocabulário com o tamanho do texto, em comparação com o modelo $v = Kn^\beta$, para os valores de K e β mostrados na Tabela 5.1. Os parâmetros desse modelo foram apontados em [2] e, como se vê pelo gráfico, ajustam-se satisfatoriamente ao tamanho real do vocabulário da coleção TREC.

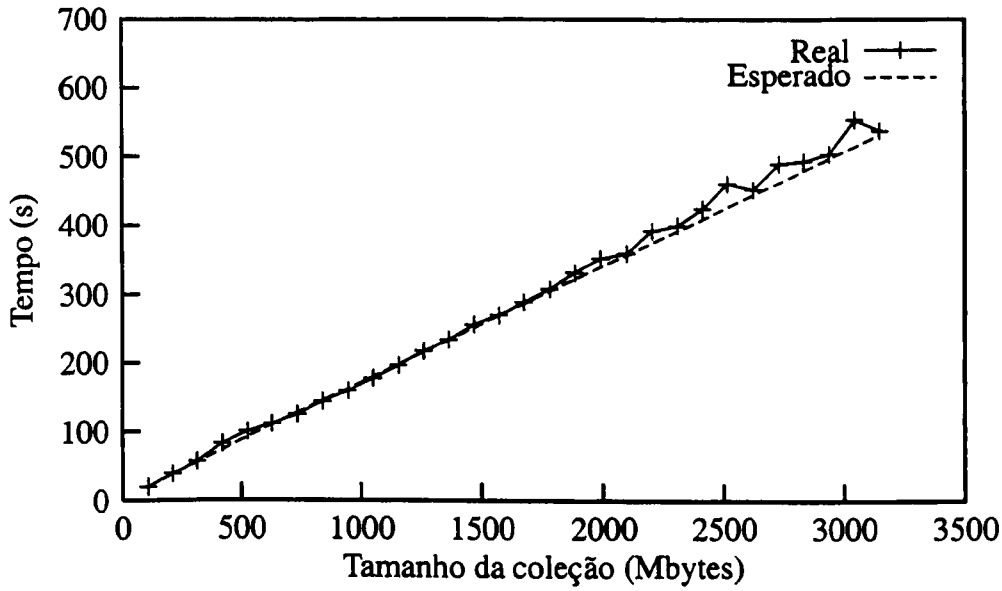


Figura 5.2: Tempo da fase de levantamento do vocabulário.

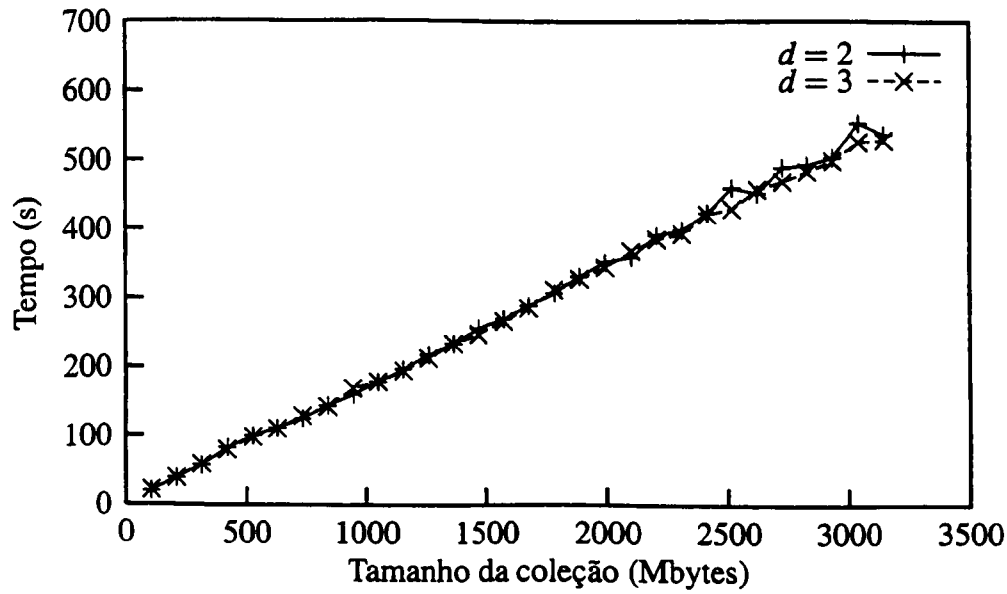


Figura 5.3: Tempo de levantamento do vocabulário utilizando-se grafos de 2 e 3 dimensões.

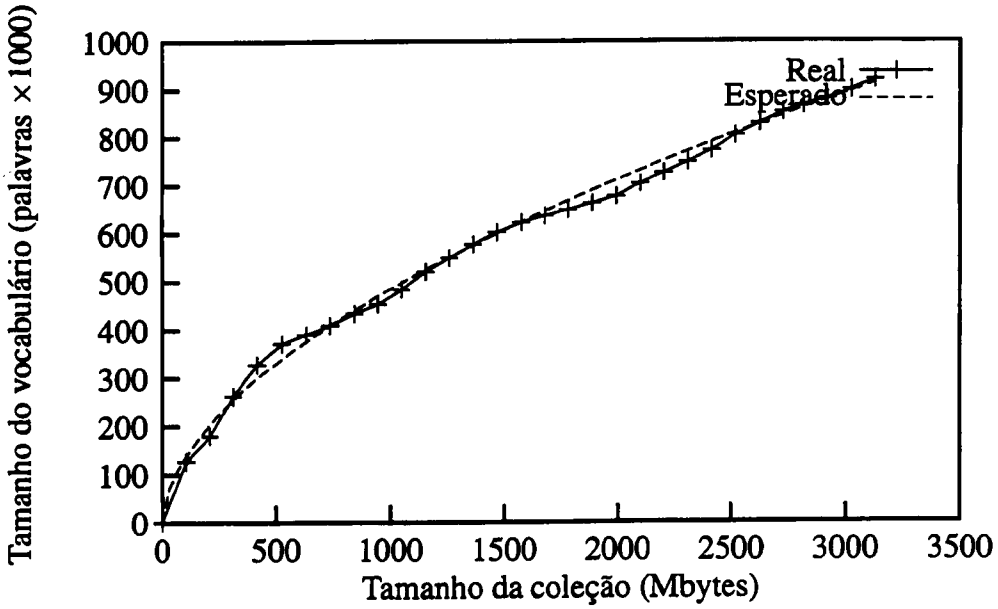
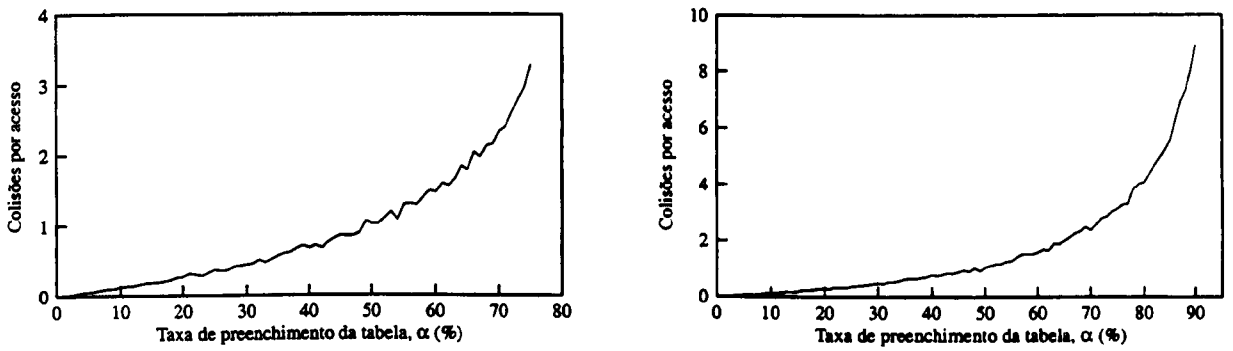


Figura 5.4: Crescimento do vocabulário com o tamanho do texto.

Como citado na Seção 2.2.2, a taxa de preenchimento da tabela *hash* normal é mantida entre 25% e 75% para garantir um pequeno número de colisões. Isso é ilustrado pelos gráficos da Figura 5.5. Em (a), mostra-se o número médio de colisões por acesso à tabela para cada percentagem de preenchimento, até 75%, como feito pelo sistema **DIG**. Nota-se que o número de colisões mantém-se baixo, ficando abaixo de 1 para taxas inferiores a 50% e chegando a um máximo de 3 para 75%. Já (b) mostra o número de colisões quando permite-se a taxa de preenchimento chegar a 90%. Percebe-se que, para cada inserção, pode-se ter até 9 colisões quando se aproxima do preechimento máximo, o que leva a uma significativa degradação no desempenho da tabela e deve, portanto, ser evitado.

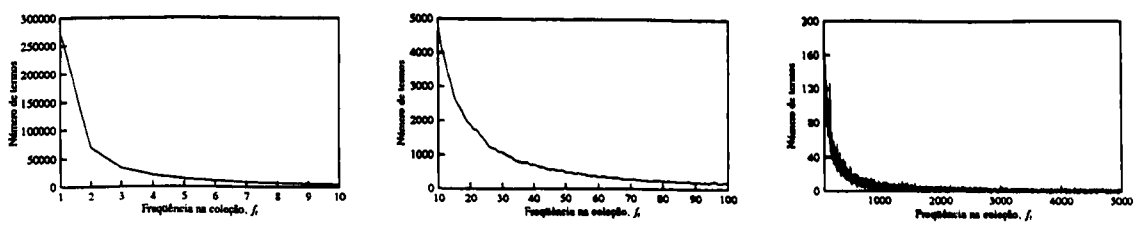
Uma importante propriedade da distribuição de termos no vocabulário da coleção é ilustrada pela Figura 5.6. Ela mostra o número de termos da coleção TREC que possuem cada uma das freqüências f_t listadas no eixo X. A freqüência f_t indica o número de documentos da coleção nos quais o termo ocorre, isto é, o tamanho da sua lista invertida. Como se vê, essa distribuição é extremamente tendenciosa e o gráfico teve que ser quebrado em três partes para que ele pudesse ser diferenciado dos eixos.

Mais de 250.000 dos 955.734 termos distintos aparecem em apenas um dos 1.660.974 documentos da coleção, número que cai para cerca de 70.000 para os termos que surgem em exatamente dois documentos. Termos que ocorrem em três documentos já ficam em cerca de 30.000 e esse valor continua decrescendo, de forma mais lenta, à medida que se tomam freqüências mais altas. Para $f_t = 50$ têm-se menos que 500 termos enquanto que, para $f_t = 500$, cerca de 15 termos e, para $f_t = 1000$, cerca de 10. O termo mais freqüênte, "the", aparece em 1.055.810 documentos, ou seja, em mais de 63% da coleção.



(a) $\alpha_{max} = 0.75$ (b) $\alpha_{max} = 0.90$

Figura 5.5: Média de colisões em acessos à tabela *hash* em função de seu preenchimento.



(a) $f_t = 1,\ldots,10$ (b) $f_t = 10,\ldots,100$ (c) $f_t = 100,\ldots,5000$

Figura 5.6: Número de termos para cada valor de freqüência na coleção, f_t.

5.3.2 Geração do índice

Nesta seção apresentam-se os gráficos relativos à execução da fase de geração seqüencial de índices. Os gráficos que validam as análises de tempo de execução são apresentados em primeiro lugar, seguidos de figuras ilustrando outras propriedades relacionadas.

Validação do modelo

A Figura 5.7 mostra o tempo de execução da fase de geração do índice invertido, juntamente com a curva predita pelo modelo, quando se varia o tamanho da entrada. A memória utilizada pelas execuções ilustradas é 64 Mbytes. Percebe-se o ajuste satisfatório do modelo aos pontos reais, bem como o comportamento claramente linear do tempo de execução em relação ao tamanho da entrada. Esse comportamento não corresponde à análise de complexidade do algoritmo, que indica ser ele $O(n\log n)$. Na equação a seguir, como $R \propto n/M$ e M é fixo, tem-se $R = O(n)$.

$$\begin{aligned}
t_{SEQ_A} &= O(n) \qquad \text{(como deduzido na seção anterior)} \\
t_{SEQ_B} &= n(t_r + t'_p) + \lambda x t_{ct} + f'(t_z + t_r) = O(n) \\
t_{SEQ_C} &= f'\left(\frac{R-1}{RM} t_s + t_r + t'_z\right) + x\lceil \log R \rceil t_{ct} + f(t_r + t_z) \\
&= O(n)O((n-1)/n) + O(n)O(\log n) + O(n) = O(n\log n).
\end{aligned}$$

Como se vê pela análise, o único termo em $O(n\log n)$, correspondente à ordenação das triplas pelo *heap*, durante a intercalação. Essa etapa não se destaca das demais, pelo fato das operações nela realizadas (simples comparações de triplas) serem muito rápidas, de forma que o tempo de execução é dominado pelas demais fases, lineares com o tamanho da entrada.

A comparação do tempo de execução real e esperado quando se varia a memória disponível para a indexação é feita pela Figura 5.8. O modelo prevê que pouca variação ocorrerá com a mudança na memória disponível, sobretudo devido às otimizações aplicadas sobre a fase de intercalação dos *runs*, a única sensível ao tamanho da memória. Com isso, as demais fases, independentes da quantidade de memória disponível, dominam o tempo de execução e fazem com que ele seja praticamente constante. A dependência da fase de intercalação com a memória disponível é analisada a seguir, lembrando-se que $R \propto n/M$ e n é fixo, logo, $R = O(1/M)$.

$$\begin{aligned}
t_{SEQ_C} &= f'\left(\frac{R-1}{RM} t_s + t_r + t'_z\right) + x\lceil \log R \rceil t_{ct} + f(t_r + t_z) \\
&= O(1) + O\left(\log \frac{1}{M}\right) + O(1)
\end{aligned}$$

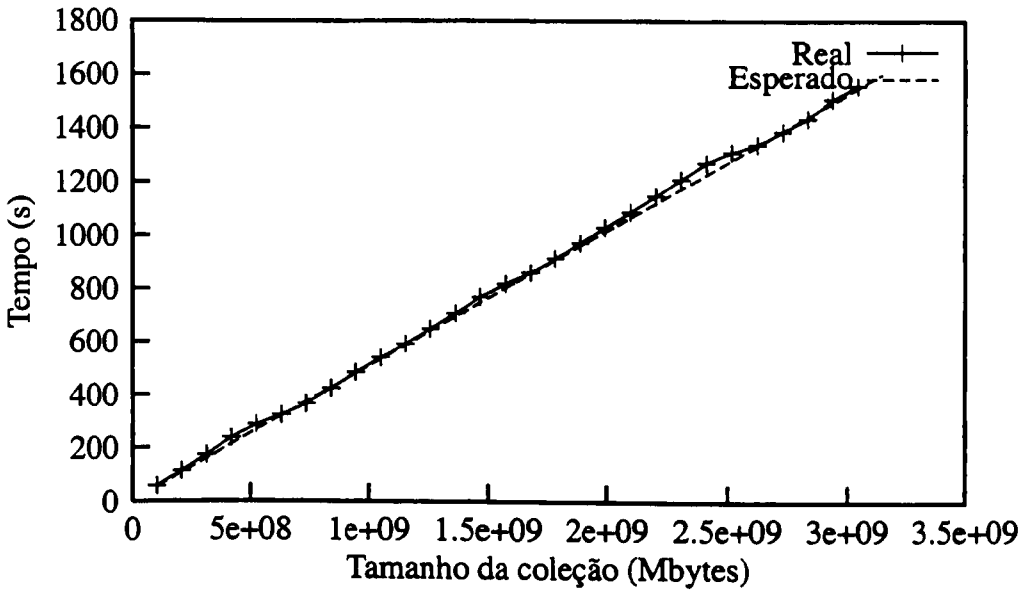


Figura 5.7: Tempo da fase de indexação, por tamanho da coleção.

Portanto, o tempo de execução tende a apresentar uma leve queda à medida que a memória disponível aumenta. Isso pode ser notado pelo formato abaulado da curva "Real" na Figura 5.8. O tempo de execução volta a subir logo após pois, com o maior consumo de memória pelo processo, aumenta a probabilidade das páginas de seu endereçamento serem jogadas para disco.

Somando-se o tempo de execução da fase de geração do índice ao do levantamento do vocabulário, tem-se o tempo total da indexação seqüencial. Para os 3 gigabytes da coleção TREC, esse valor corresponde a 2134 segundos, ou cerca de $35\frac{1}{2}$ minutos.

Outros gráficos

A seguir são apresentados outros gráficos relativos à execução da fase de geração de índices. A Figura 5.9 compara os tempos de indexação quando se utiliza a ordenação linear e o *Quicksort*. Notam-se claramente os ganhos em tempo de execução, que pode ser reduzido até pela metade empregando-se a ordenação linear.

Os tempos de indexação com e sem o uso de compressão são mostrados na Figura 5.10. Esse gráfico mostra que o uso de compressão de dados propicia não só economias em espaço ocupado pelos índices, mas também ganhos em tempo de execução, devido à redução do volume de dados transferidos entre memória e disco. Além disso, a indexação sem compressão não pode executar para toda a TREC, pois o arquivo temporário ultrapassa 2 Gbytes, um limite superior do sistema Linux e das arquiteturas de 32 bits em geral. Por reduzir o tamanho dos índices, o uso de compressão permite a indexação de entradas maiores antes de se atingir esse limite. Para ser possível indexar textos muito grandes (superiores a 20 Gbytes, no caso da execução com

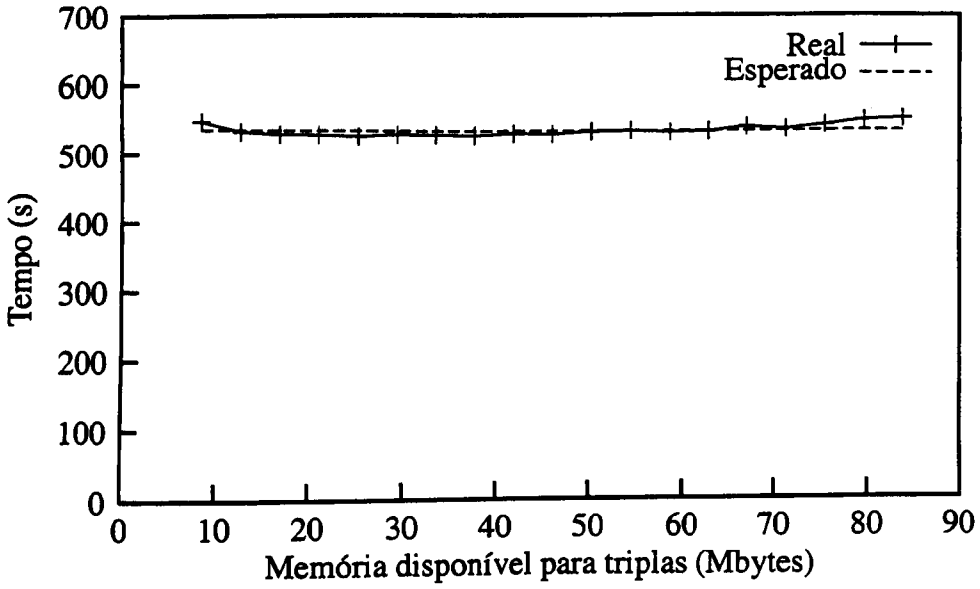


Figura 5.8: Tempo da fase de indexação, por memória disponível.

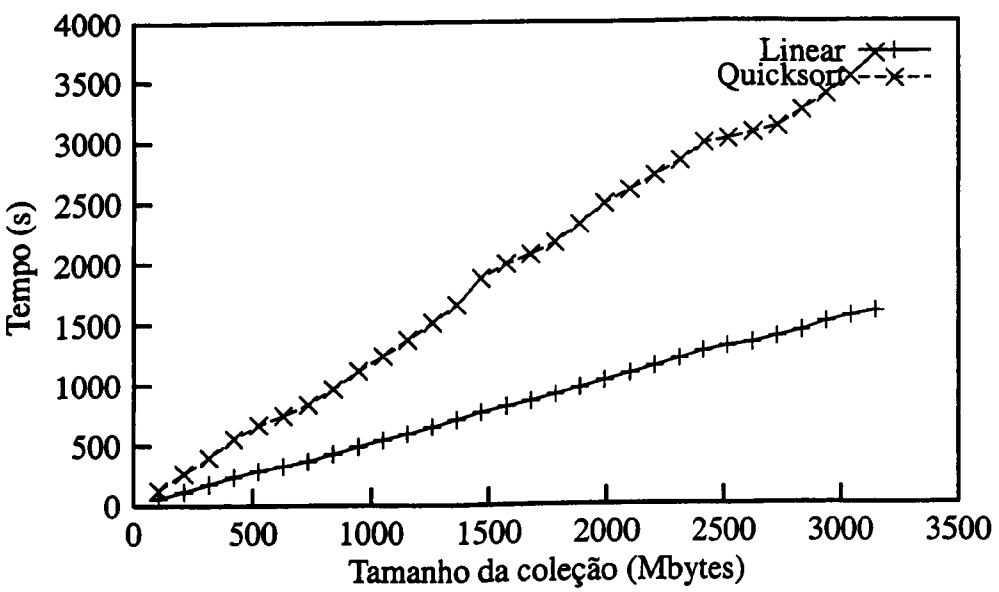


Figura 5.9: Tempo de indexação para a ordenação linear e o *Quicksort*.

compressão), bastaria permitir a criação de mais de um arquivo de índice.

Na Figura 5.11 comparam-se os tempos de indexação quando se empregam funções *hash* perfeitas geradas por grafos de 2 e 3 dimensões. Nota-se que, apesar das funções geradas por trigrafos consumirem menos memória auxiliar, a sobrecarga representada pela computação de uma dimensão a mais – correspondente a aplicar um vetor aleatório a mais para cada palavra do texto – torna essa alternativa menos vantajosa. Isso mostra que o *parsing* é uma atividade crítica em relação ao desempenho da indexação, fator que acentua a característica linear dos algoritmos, uma vez que o tempo de *parsing* é diretamente proporcional ao tamanho do texto.

A Figura 5.12 mostra o tamanho dos índices invertidos temporário e final, comprimidos, em função do crescimento do texto. Como mostra a figura, o índice temporário corresponde a cerca de 10% do tamanho do texto, ao passo que o índice final equivale a 9% do mesmo. Por outro lado, sem o uso de compressão, o índice temporário pode chegar a 100% do texto e o final, a 60%. Como mostra a Tabela 5.1, o número de bits tomado por cada tripla é, em média, 11.8 no arquivo temporário e 10.5 no arquivo final. Isso evidencia a economia proporcionada em relação aos 96 bits e 64 bits ocupados por cada tripla nos arquivos temporário e final, respectivamente, quando não se utiliza compressão.

A Figura 5.13 ilustra outra propriedade importante de arquivos invertidos. Ela destaca o número de triplas no índice final cujos componentes $f_{t,d}$ correspondem a cada uma das freqüências mostradas no eixo X. O índice foi construído sobre 1 Gbyte da coleção TREC. Esse histograma explicita a distribuição das freqüências relativas $f_{t,d}$ dentro das listas invertidas, mostrando ser ela também bastante tendenciosa; caso ele não fosse dividido em três partes, seria impossível diferenciá-lo dos eixos.

Nota-se que mais de 60 milhões de triplas possuem o componente $f_{t,d}$ igual a 1, valor que cai para cerca de 13 milhões quando se toma a freqüência 2. Há ainda mais de 700 triplas com freqüência igual a 100. Para valores elevados de $f_{t,d}$, o que provavelmente ocorre para termos muito freqüentes em documentos longos, o número de triplas varia consideravelmente de uma freqüência para outra, mas ainda é considerável – para a freqüência 500, há 28 termos, enquanto que para 1000, há ainda 42 triplas no arquivo invertido.

Finalmente, a Figura 5.14 apresenta outra análise comparando as ordenações pelo método linear e pelo *Quicksort*. Em (a) são mostrados os tempos de ordenação para ambos os métodos, destacando a grande diferença de eficiência entre eles. O gráfico (b) inclui o tempo da ordenação linear e a modelagem $\lambda x t_{ct}$, com λ e ct dados pela Tabela 5.2. No gráfico (c) é apresentado o tempo de execução do *Quicksort*, ao lado da modelagem criada para ele. Para isso realizou-se uma interpolação polinomial e o resultado que proporcionou o melhor coeficiente de correlação ($R^2 = 0.9935$) foi um polinômio de grau 2, mostrando que o *Quicksort* realmente se aproxima do comportamento quadrático quando é fornececida uma entrada de dados tendenciosa, como é a distribuição de Zipf. Os coeficientes obtidos para o polinômio $ax^2 + bx + c$ foram $a = 5.55 \times 10^{-4}$, $b = 1.17 \times 10^3$ e $c = 2.52 \times 10^9$.

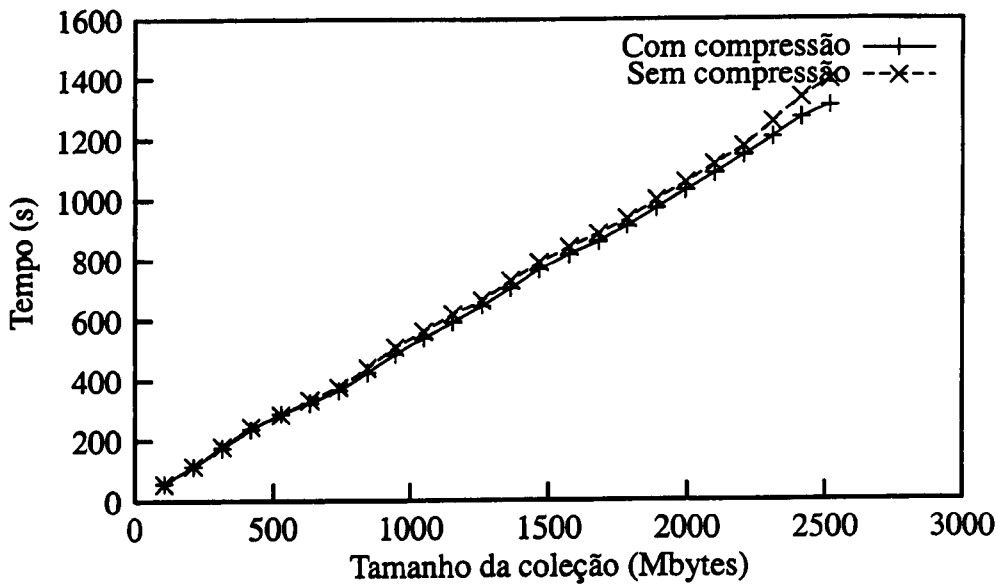


Figura 5.10: Tempo de indexação com e sem o uso de compressão.

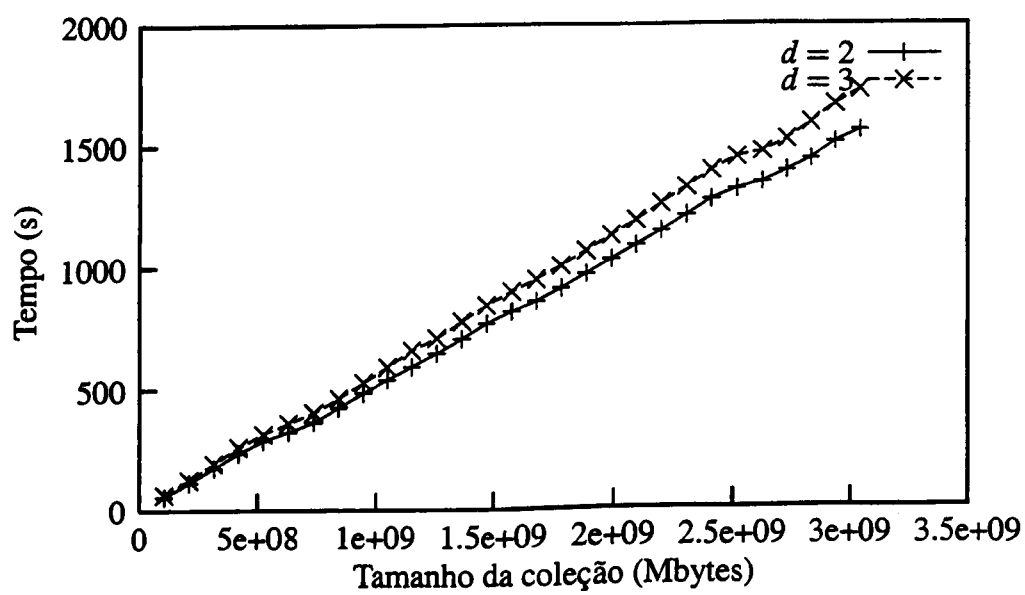
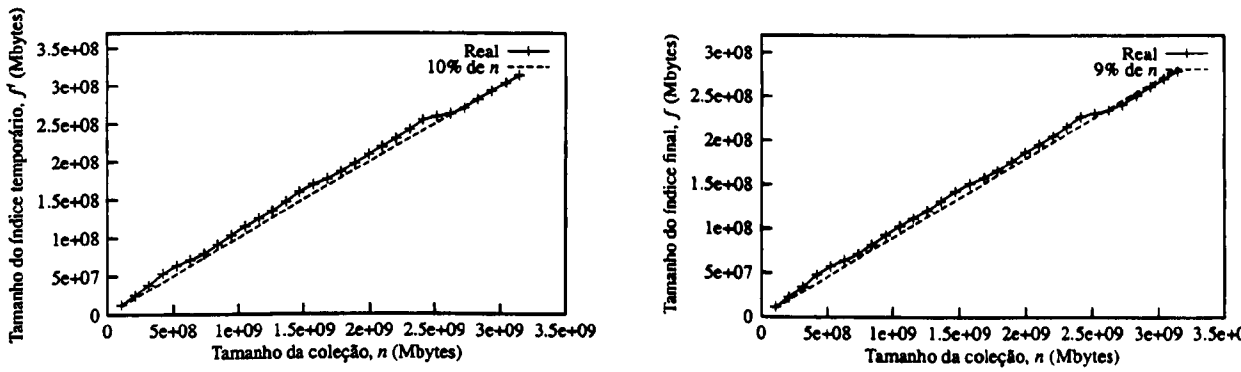
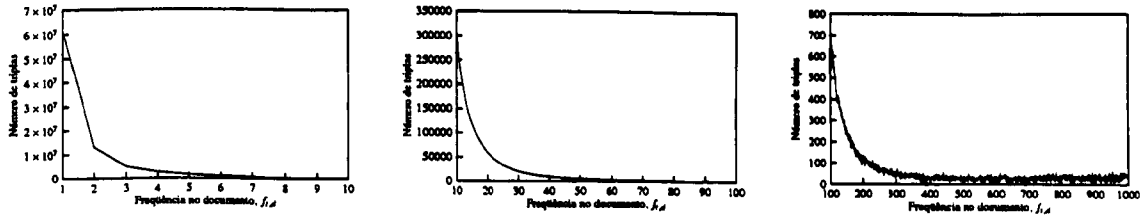


Figura 5.11: Tempo de indexação para funções *hash* de duas e três dimensões.



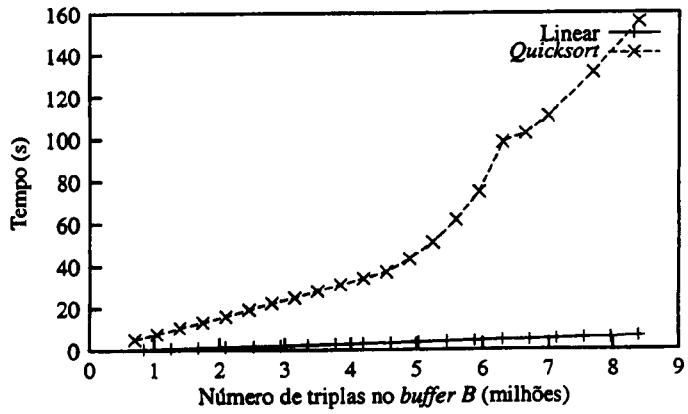
(a) Índice temporário (b) Índice final

Figura 5.12: Tamanhos dos índices temporário e final em função do tamanho da coleção.

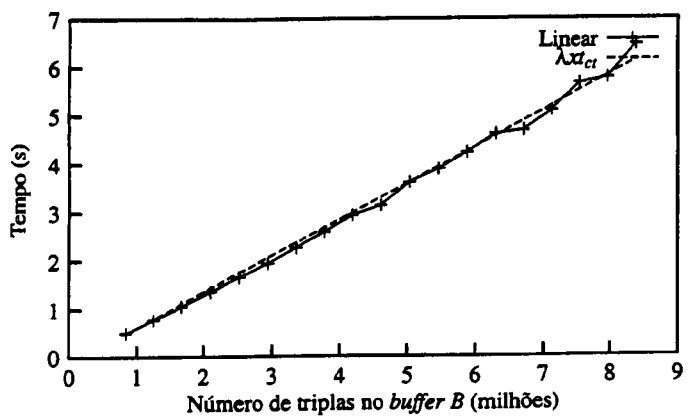


(a) $f_{t,d} = 1, \ldots, 10$ (b) $f_{t,d} = 10, \ldots, 100$ (c) $f_{t,d} = 100, \ldots, 1000$

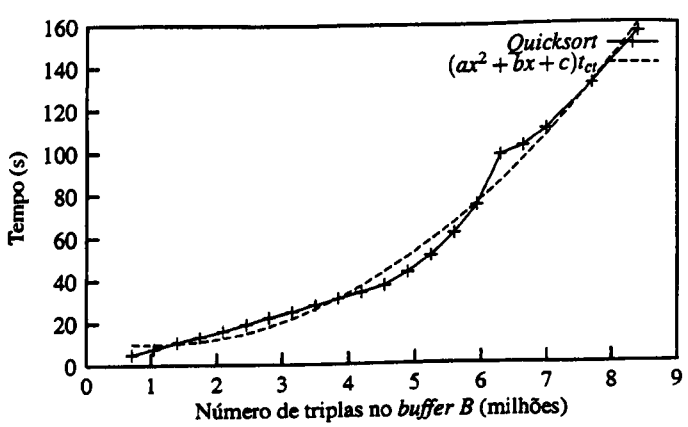
Figura 5.13: Número de triplas para cada valor de freqüência em documentos, $f_{t,d}$.



(a) Linear × *Quicksort*



(b) Linear × $\lambda x t_{ct}$



(c) *Quicksort* × $(ax^2 + bx + c)t_{ct}$

Figura 5.14: Tempos de ordenação das triplas pelos métodos linear e *Quicksort*.

5.4 Geração Distribuída

Os resultados obtidos para os algoritmos distribuídos de indexação são aqui apresentados. Incluem-se gráficos para a verificação dos modelos criados em relação aos resultados reais, bem como para a comparação dos algoritmos entre si. Comparam-se também os algoritmos distribuídos ao algoritmo seqüencial, analisando-se o *speedup* obtido.

5.4.1 Obtenção do vocabulário

Nesta seção procura-se validar a expressão de tempo para a fase de levantamento do vocabulário global. Apresenta-se a variação do tempo de execução em função do tamanho do texto e do número de processadores. Analisa-se também o *speedup* esperado e real utilizando-se 2, 3 e 4 processadores.

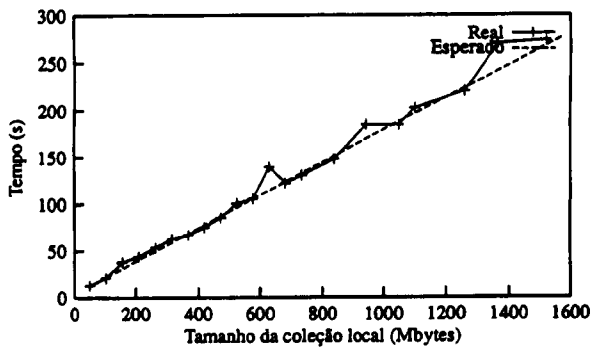
Validação do modelo

A Figura 5.15 traz, na coluna da esquerda, os gráficos do tempo de execução da fase de levantamento do vocabulário contra os valores estimados pelo modelo, para 2, 3 e 4 processadores. A geração da função *hash* perfeita utiliza grafos bidimensionais. Nota-se que o modelo aproxima satisfatoriamente o tempo de execução para a maioria dos tamanhos de entrada, falhando apenas nos casos em que há uma variação inesperada no tempo real.

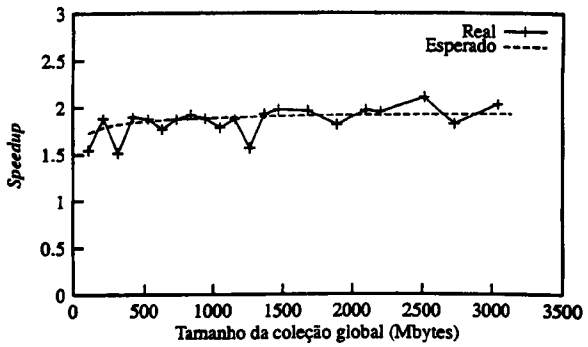
Destaca-se pelo gráfico a maior instabilidade na geração da função *hash* perfeita, que cresce com o número de processadores. Isso deve-se ao fato dessa geração ser feita seqüencialmente no processador 0, ao passo que a função é relativa ao vocabulário da coleção como um todo. Como a proporção do número de termos no vocabulário em relação ao texto local aumenta à medida que mais processadores são empregados, as variações tornam-se mais visíveis.

O *speedup* proporcionado pela execução distribuída da fase de obtenção do vocabulário, para 2, 3 e 4 processadores, é mostrado na coluna da direita da Figura 5.15, em função do tamanho do texto. Juntamente apresenta-se o *speedup* esperado, baseado no modelo analítico. Os gráficos apresentam no eixo X o tamanho em bytes da coleção global processada e, no eixo Y, o valor t_{SEQ_A}/t_{GV}, isto é, a razão entre o tempo de levantamento do vocabulário em uma única máquina e o tempo executando-se a mesma tarefa em p máquinas.

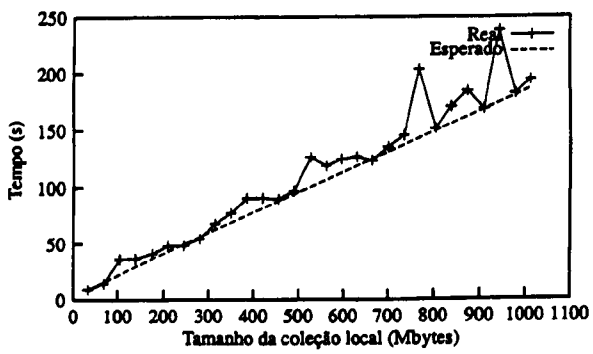
Observa-se que o *speedup* se mantém próximo do número de processadores empregados para praticamente todos os tamanhos de entrada, mostrando um bom aproveitamento do paralelismo. Realmente, as etapas da execução cuja complexidade é linear com o tamanho do texto, isto é, leitura e *parsing*, são executadas em paralelo, sem nenhuma comunicação entre as máquinas. Os trechos seqüenciais da execução são dependentes do tamanho do vocabulário, que cresce sublinearmente com o texto. Assim, o *speedup* é alto e aumenta com o tamanho da entrada, até atingir se chegar ao ponto no qual o crescimento do vocabulário não é mais significativo.



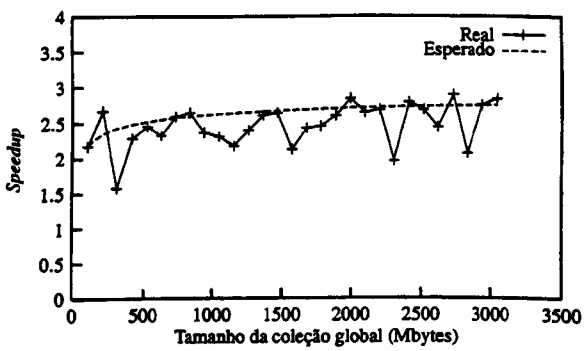
(a) Tempo de execução para $p = 2$



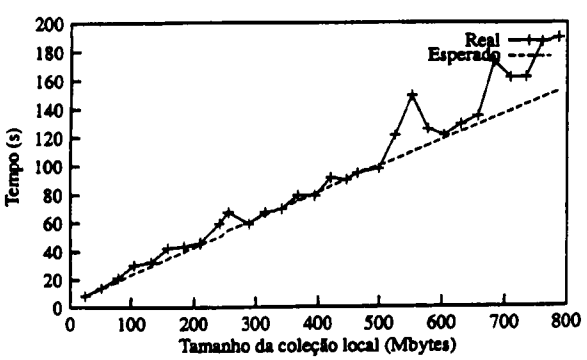
(b) *Speedup* para $p = 2$



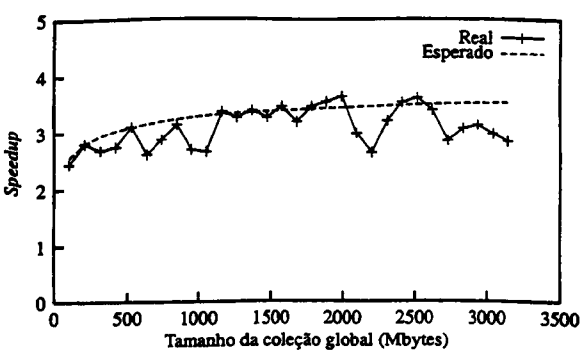
(c) Tempo de execução para $p = 3$



(d) *Speedup* para $p = 3$



(e) Tempo de execução para $p = 4$



(f) *Speedup* para $p = 4$

Figura 5.15: Validação do tempo de execução e *speedup* para a obtenção do vocabulário.

5.4.2 Geração do índice

Apresentam-se nesta seção os resultados relativos à fase de geração do índice para os três algoritmos distribuídos. Procuram-se validar as análises de tempo de execução em função do tamanho do texto local, do número de processadores e da memória disponível por processador. Incluem-se também os gráficos de *speedup* esperado e real para cada um dos algoritmos, em função do tamanho do texto.

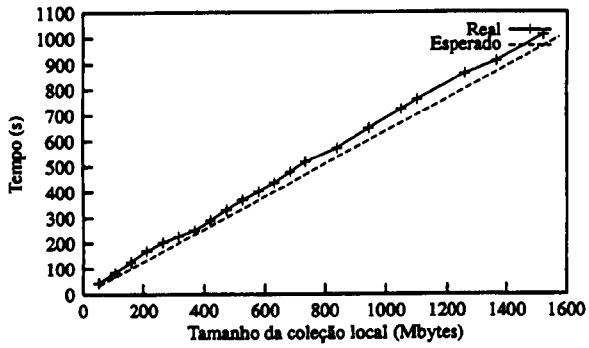
Validação do modelo

As Figuras 5.16, 5.17 e 5.18 mostram os tempos de execução dos algoritmos distribuídos, bem como seus respectivos *speedups*, em função do tamanho do texto. Para esses experimentos, foram utilizados 32 Mbytes de memória para as triplas nos algoritmos LL e LR, ao passo que o algoritmo RR utilizou 16 Mbytes. Esses valores possibilitaram um melhor desempenho, como destacado pelas Figuras 5.19 e 5.21.

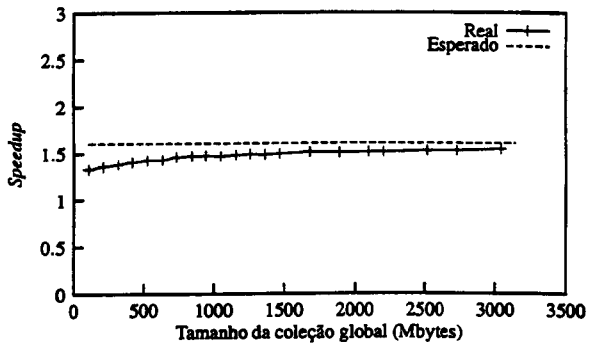
A coluna esquerda da Figura 5.16 compara o tempo de execução do algoritmo LL com os valores preditos pelo modelo, utilizando-se 2, 3 e 4 processadores. Nota-se que a aproximação é bastante satisfatória para 2 e 3 processadores, ao passo que, para 4 máquinas, há uma discrepância de cerca de 30% entre o tempo estimado e o real. Esse fenômeno pode ser atribuído ao aumento da concorrência entre as *threads* executando em cada máquina. De acordo com a implementação descrita na Seção 4.5, há $p+1$ *threads* ativas por máquina: uma *thread* principal processando o texto local, uma *thread* de envio de triplas e $p-1$ para a recepção.

Aumentando-se o número de máquinas participantes de indexação, cresce também a competição por recursos em cada máquina, devido ao maior número de processos ativos. Apesar do processamento mais pesado ser realizado pela *thread* principal, as demais *threads* também necessitam tempo de CPU e disco e, logo, há mais trocas de contexto. Esse efeito é ainda mais acentuado pelo fato de *multithreading* no sistema Linux não ser implementado no *kernel*, mas sim emulado através de processos comuns e mapeamento de memória, fazendo com que o chaveamento seja mais caro.

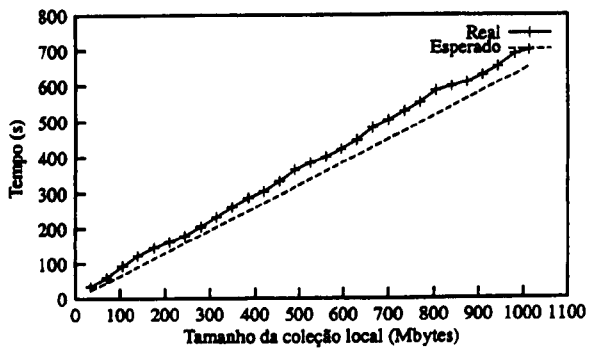
O modelo falha em prever esse fenômeno na medida em que não representa o custo de recepção de triplas, somente de seu envio, como já citado no Capítulo 3. Entretanto, representar esse custo não é simples, pois deve considerar detalhes do sistema operacional, como sua eficiência para lidar com *multithreading* e compartilhamento de recursos. Para corrigi-la, é preciso dividir o tempo de execução da fase na qual as *threads* concorrem por recursos (no caso do algoritmo LL, a fase C, para os demais, a fase B), pela percentagem de utilização desses recursos pela *thread* principal, da forma descrita em [50]. Como a utilização é um valor menor que 1, o tempo obtido seria maior que o original, representando o tempo gasto com a execução das *threads* auxiliares e trocas de contexto. Apesar dessa inacurácia, o modelo acerta em prever que o comportamento do algoritmo LL é predominantemente linear com o tamanho da entrada.



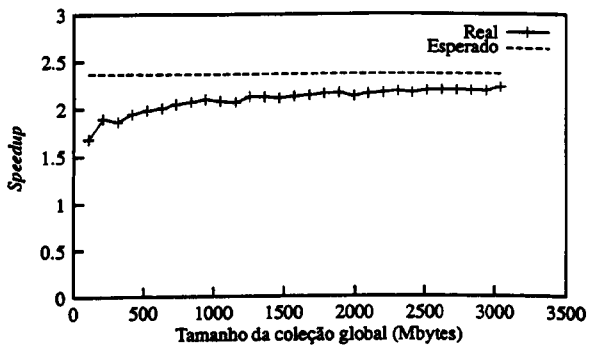
(a) Tempo de execução para $p = 2$



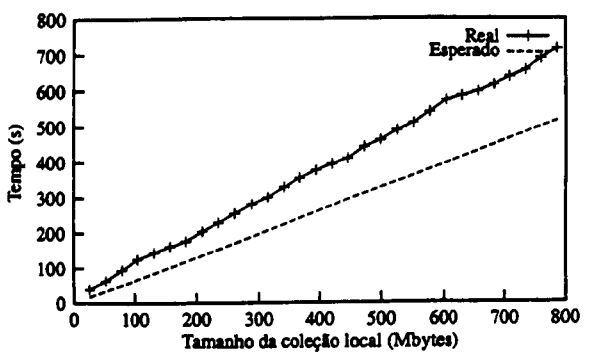
(b) *Speedup* para $p = 2$



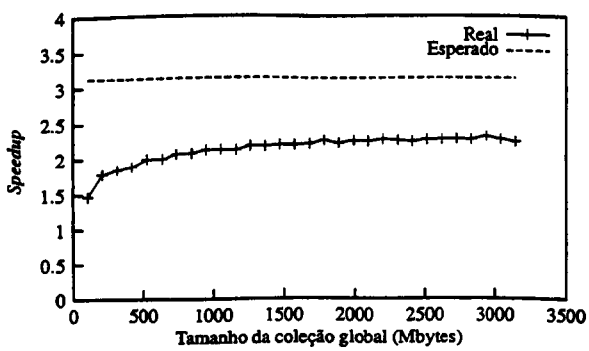
(c) Tempo de execução para $p = 3$



(d) *Speedup* para $p = 3$



(e) Tempo de execução para $p = 4$



(f) *Speedup* para $p = 4$

Figura 5.16: Validação do tempo de execução e *speedup* para o algoritmo LL.

O efeito notado para o algoritmo LL mostra-se ainda mais acentuado para o algoritmo LR, como se vê pela Figura 5.17. Isso ocorre porque, no algoritmo LR, a competição entre as *threads* é maior, pois ocorre durante toda a indexação, sempre que o *buffer* de uma máquina torna-se cheio e triplas são enviadas pela rede. Isso aumenta a probabilidade de que uma das *threads* não obtenha acesso à CPU ou até mesmo sofra um *swap* para disco, mesmo que esteja pronta para executar. No algoritmo LL, a troca de ocorrências só ocorre durante a intercalação, uma fase mais curta, o que faz com que haja menos competição.

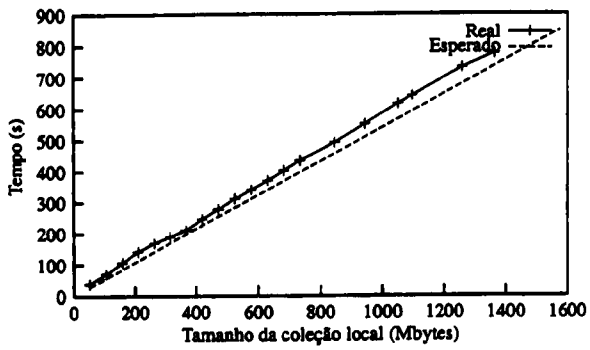
O gráfico para o tempo de indexação pelo algoritmo LR mostra-se retilíneo para todos os números de máquinas utilizados, destacando seu comportamento linear. Para $p = 4$, realizou-se uma regressão linear sobre os dados, identificando-se a reta $y = ax + b$ que melhor se ajusta a eles. Os coeficientes obtidos foram $a = 7.98 \times 10^{-7}$ e $b = 18.38$, para uma correlação dada por $R^2 = 0.9995$. Com esses valores, pode-se prever que, no mesmo ambiente de execução empregado nos testes, isto é, PCs comuns ligados por uma rede rápida, é possível indexar uma coleção de 100 Gbytes (como é o caso da TREC-7) em menos de 6 horas.

Para o algoritmo RR, o modelo se mostra satisfatório quando se utilizam 2 e 3 processadores, segundo ilustrado pelas duas primeiras linhas da Figura 5.18. Entretanto, para 4 máquinas, a sobrecarga representada pela excessiva concorrência entre as *threads* degrada o desempenho a tal ponto que o tempo de execução perde seu comportamento linear. O algoritmo RR é o único para o qual as *threads* concorrem não só pela CPU e outros recursos da máquina, mas também por uma estrutura de dados em memória, o arranjo de triplas. A necessidade de sincronização das *threads* no acesso a essa área compartilhada faz com que o algoritmo RR não seja escalável para um número maior de processadores. Essa característica é ainda mais acentuada pelo fato do sistema Linux implementar compartilhamento de memória entre *threads* através do simples mapeamento de memória entre processos.

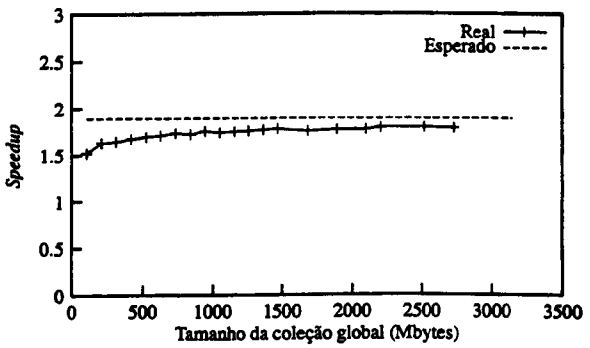
Ainda em respeito à validação do modelo, a Figura 5.19 apresenta os gráficos de tempo de execução em função da memória disponível para as triplas. Nota-se que o algoritmo LL é o menos sensível à variação da memória, pois é o menos sujeito à concorrência das *threads*.

Já o algoritmo LR pode sofrer maiores variações no tempo de execução caso se aumente muito a memória destinada ao arranjo de triplas, pois as trocas de contexto para ele são mais intensas e a área do arranjo tem maiores chances de ser removida da memória principal. Observa-se também que o algoritmo LR atinge seus menores tempos de execução com quantidades de memória mais elevadas que os demais. Isso deve-se a sua intercalação, que possui $p \times R$ caminhos e depende de mais memória para a redução do número de *seeks* a disco.

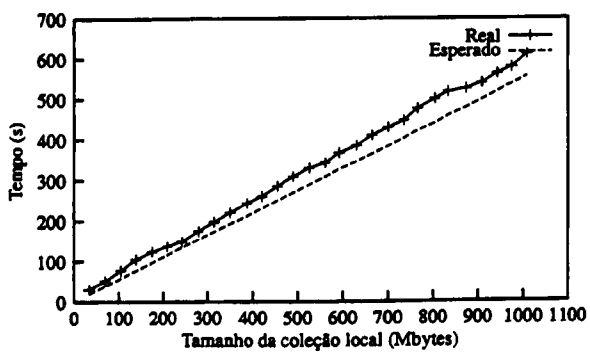
Por fim, o algoritmo RR mostra-se o mais sensível à variação da memória, por ser aquele no qual a concorrência entre as *threads* é mais acentuada e pelo fato do arranjo de triplas ser uma estrutura central em seu desempenho. Caso o arranjo ocupe uma área muito grande, aumenta a probabilidade de que parte dessa área seja removida da memória principal, o que pode fazer com que várias *threads*, em vez de apenas uma, gerem faltas de páginas.



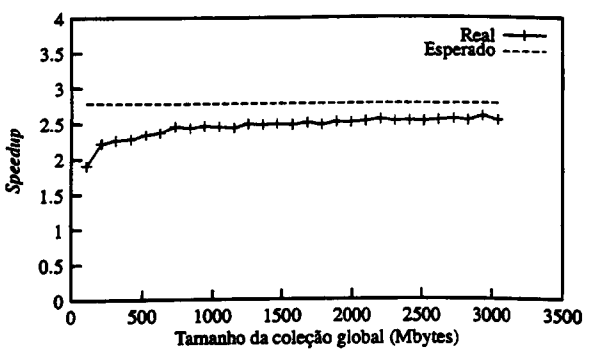
(a) Tempo de execução para $p = 2$



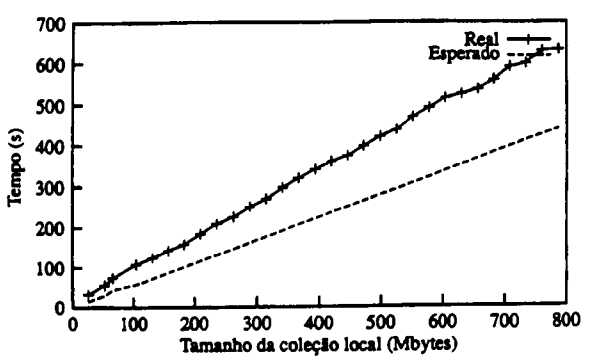
(b) *Speedup* para $p = 2$



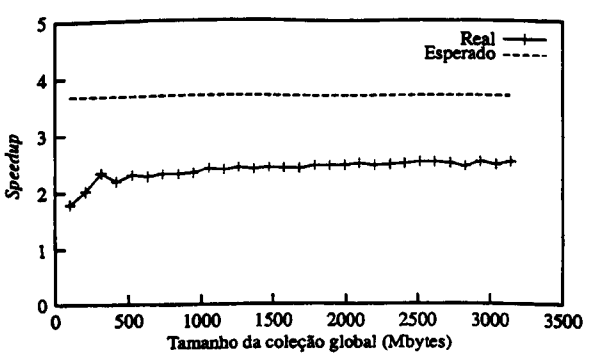
(c) Tempo de execução para $p = 3$



(d) *Speedup* para $p = 3$

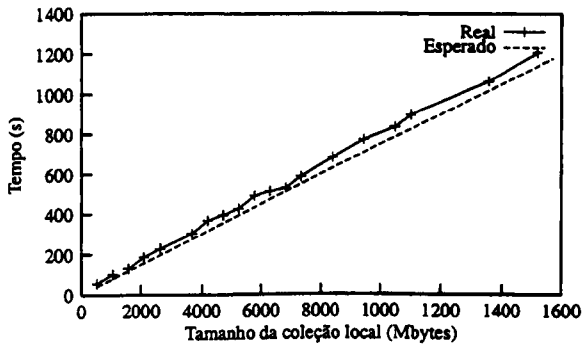


(e) Tempo de execução para $p = 4$

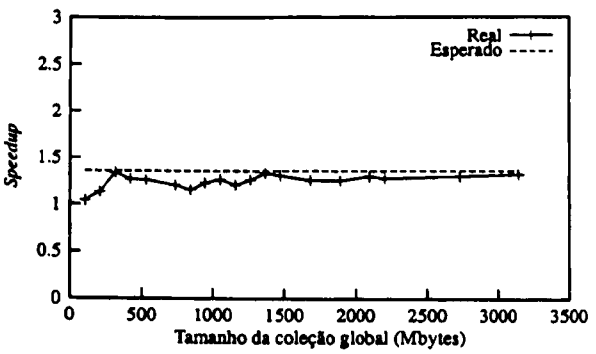


(f) *Speedup* para $p = 4$

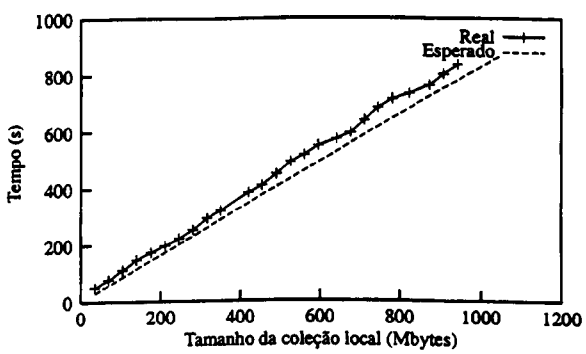
Figura 5.17: Validação do tempo de execução e *speedup* para o algoritmo LR.



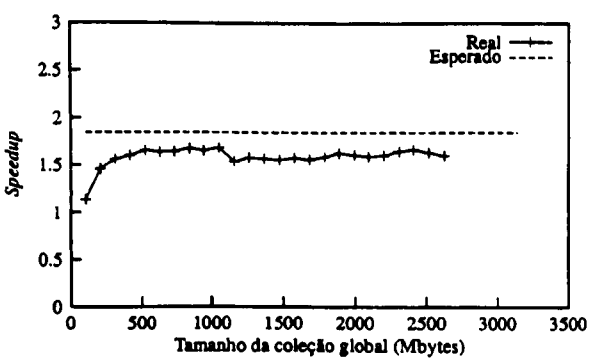
(a) Tempo de execução para $p = 2$



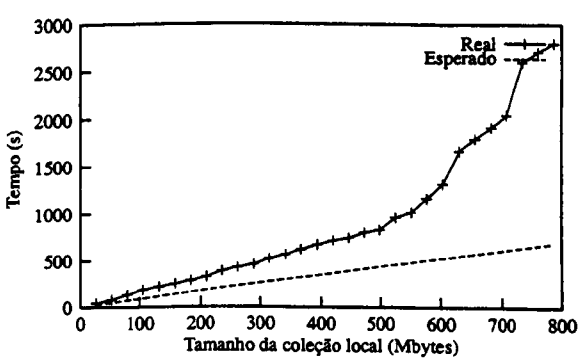
(b) *Speedup* para $p = 2$



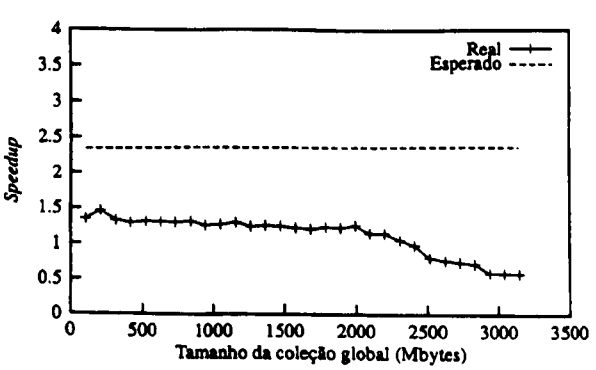
(c) Tempo de execução para $p = 3$



(d) *Speedup* para $p = 3$

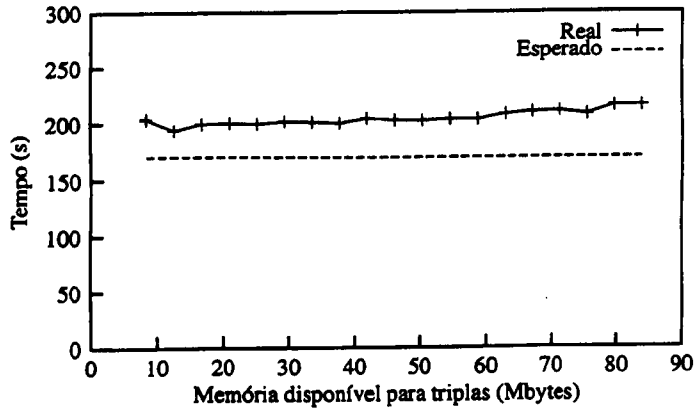


(e) Tempo de execução para $p = 4$

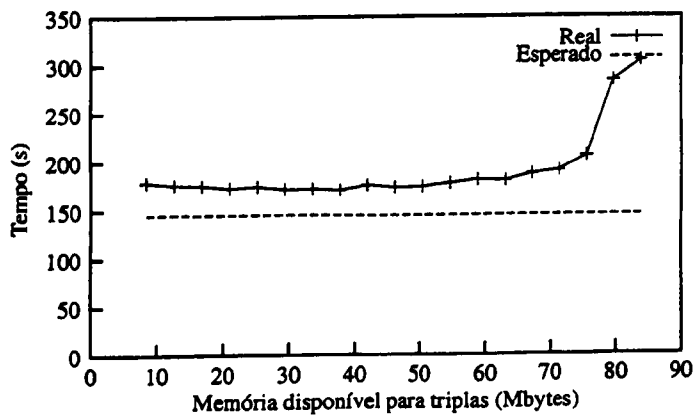


(f) *Speedup* para $p = 4$

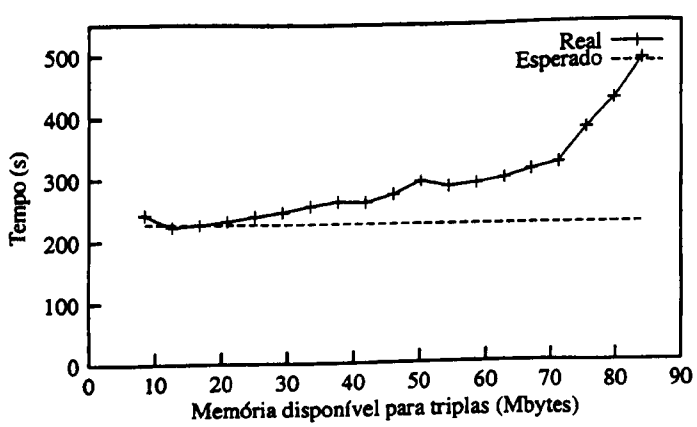
Figura 5.18: Validação do tempo de execução e *speedup* para o algoritmo RR.



(a) Algoritmo LL



(b) Algoritmo LR



(c) Algoritmo RR

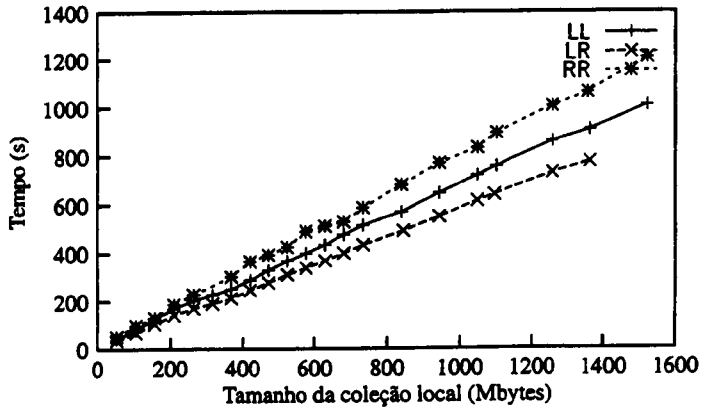
Figura 5.19: Validação do tempo da indexação distribuída em função da memória.

Outros gráficos

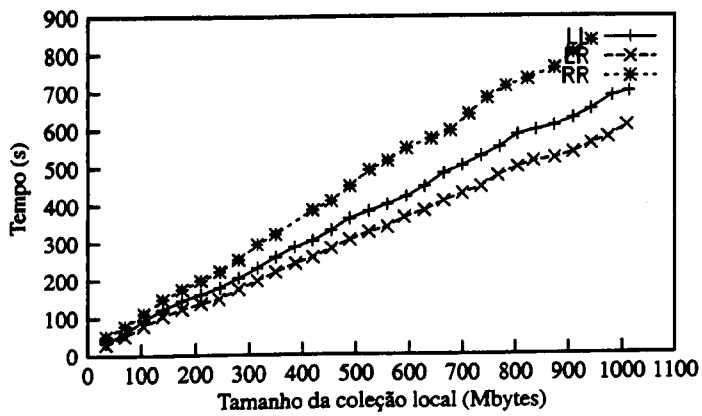
Apresentam-se agora outros gráficos comparativos dos algoritmos distribuídos. A Figura 5.20 compara diretamente o tempo de execução dos três em função do tamanho da entrada, para 2, 3 e 4 processadores. Nota-se a constante vantagem obtida pelo algoritmo LR e o déficit do algoritmo RR, que aumenta com o número de processadores.

A Figura 5.21 compara diretamente o tempo de execução dos algoritmos para um texto de 1 gigabyte, variando-se a memória disponível para as triplas. Destaca-se a rápida degradação de desempenho do algoritmo RR com o aumento excessivo dessa área de memória, seguido do algoritmo LR, ao passo que o LL mantém-se mais estável.

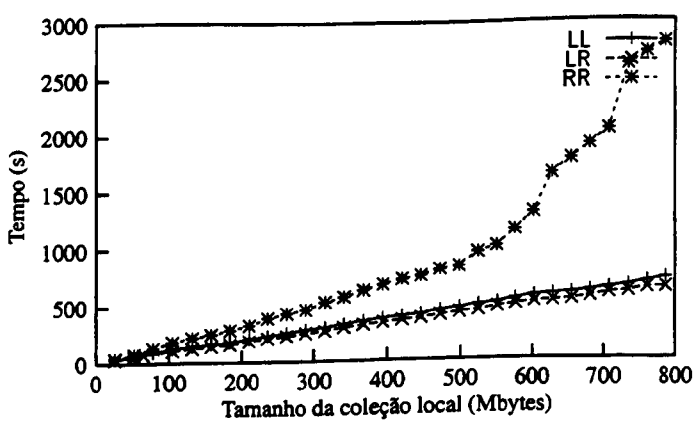
O gráfico da Figura 5.22 destaca outra interessante variável na análise dos algoritmos distribuídos, a velocidade da rede. Como não foi possível estabelecer diretamente esse valor para os experimentos, incluem-se os resultados analíticos, sem a consideração da concorrência. Nesse cenário, o número de processadores é 4. O algoritmo RR é aquele que tem seu desempenho mais afetado por redes mais lentas, pois não é capaz de comprimir os dados transmitidos. Para redes mais rápidas, o tempo de execução atinge um limite, determinado pela porção seqüencial de cada algoritmo. Essa porção é maior para o algoritmo LL, que deve executar duas intercalações, realizando uma passada a mais sobre o arquivo invertido. Os algoritmos LR e RR se emparelham nesse ponto, com uma leve vantagem para o RR, que realiza uma intercalação de R caminhos, ao passo que a intercalação do algoritmo LR possui $p \times R$ caminhos.



(a) $p = 2$



(b) $p = 3$



(c) $p = 4$

Figura 5.20: Tempo em função do tamanho da coleção para os algoritmos distribuídos.

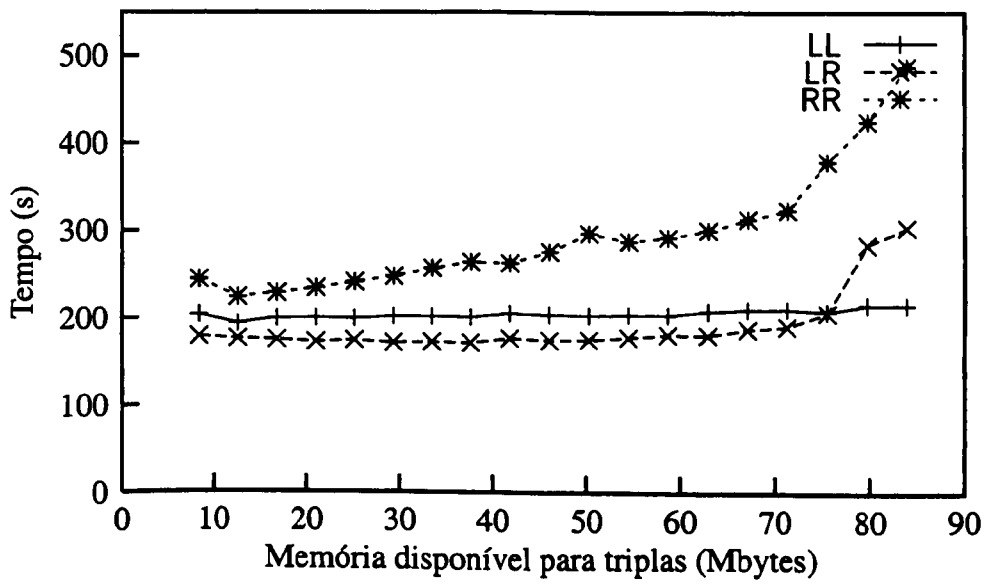


Figura 5.21: Tempo em função da memória disponível para os algoritmos distribuídos.

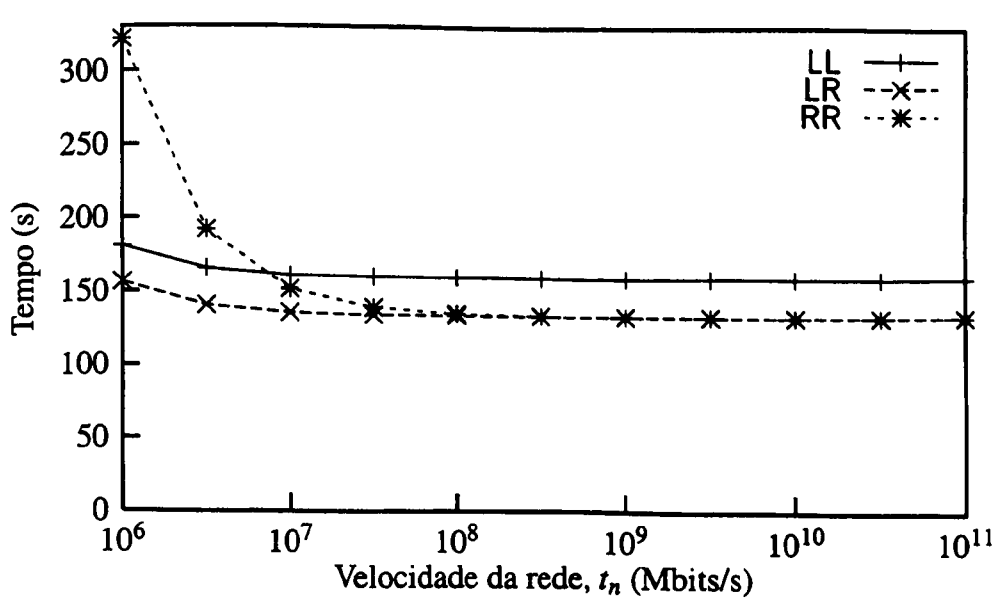


Figura 5.22: Tempo em função da velocidade da rede para os algoritmos distribuídos.

Capítulo 6

Conclusões e Trabalhos Futuros

Esta dissertação apresentou uma família de algoritmos visando a geração distribuída de arquivos invertidos para grandes coleções de texto. O ambiente de execução considerado foi uma rede rápida conectando estações de trabalho cuja memória principal pode ser consideravelmente menor que o índice gerado. Assumiu-se que as listas invertidas deveriam ser ordenadas pelas freqüências dos termos nos documentos, de forma a permitir um processamento de consultas mais rápido. As listas foram também comprimidas, para reduzir o espaço ocupado em disco e o tempo de construção.

Como base para os algoritmos distribuídos propostos, implementou-se um algoritmo de indexação seqüencial bastante eficiente. Utilizou-se para isso uma técnica linear para a ordenação de listas invertidas por freqüências, bem como estratégias para a redução do número de movimentações da cabeça de leitura do disco na fase de intercalação, acesso mais rápido ao vocabulário e *parsing* eficiente do texto.

Modelos analíticos foram elaborados para cada algoritmo proposto, sendo posteriormente calibrados e validados pelos experimentos. Estes últimos mostraram que os 3 gigabytes da coleção TREC podem ser indexados pelo algoritmo seqüencial em menos de 36 minutos, ao passo que o melhor algoritmo distribuído proposto, o LR, executando em quatro máquinas, é capaz de indexar o mesmo volume de texto em pouco mais de 13 minutos. As análises atestam que, utilizando-se as mesmas quatro máquinas, é possível inverter uma coleção de 100 gigabytes em menos de 6 horas.

Pelos resultados apresentados pode-se concluir que o processo de indexação tem um comportamento predominantemente linear em relação ao tamanho do texto processado, desde que a implementação seja cuidadosa para se reduzir a proporção das fases de complexidade $O(n \log n)$. Nota-se que as fases com o maior tempo de execução são a leitura e o *parsing* do texto, mostrando que se está próximo do limite inferior para o tempo de indexação. Isso justifica a aplicabilidade de técnicas de compressão de texto, que reduzem o volume de dados a ser lido do disco e podem permitir ganhos ainda maiores.

Também mostra-se válido o emprego de compressão sobre as listas invertidas produzidas

pela indexação, pois isso permite uma redução significativa (de 6 a 8 vezes) no espaço ocupado em disco pelos índices e ainda leva a uma redução no tempo de execução.

Outra conclusão proporcionada pelos experimentos é que o algoritmo de ordenação *Quicksort* não apresenta um bom desempenho operando sobre distribuições tendenciosas, como ocorre com listas invertidas. Nessa situação, a alternativa é utilizar métodos específicos, que explorem essas distribuições. No contexto deste trabalho foi projetada e implementada uma técnica de ordenação para listas invertidas por freqüências que possui tempo de execução linear. Para isso, ela lança mão do conhecimento das distribuições de tamanhos das listas invertidas e de freqüências dos elementos dessas listas.

O processo de intercalação realizado para a geração de arquivos invertidos também pode ser significativamente melhorado utilizando-se a memória disponível para o pré-carregamento de dados do disco e a redução do número de acessos aleatórios.

Em relação aos algoritmos distribuídos, observou-se que a competição por recursos – sejam estes estruturas de dados compartilhadas ou recursos da máquina, como a CPU – é um fator que gera uma sobrecarga significativa em aplicações de processamento intensivo, como é o caso da indexação. Mesmo sendo de difícil modelagem, essa característica não pode ser ignorada nas análises.

O algoritmo LR mostrou-se o mais eficiente dentre os três, ao passo que o RR, devido à excessiva sincronização e à incapacidade de compressão dos dados transmitidos pela rede, acabou por resultar no algoritmo mais lento e menos escalável. O algoritmo LL é escalável e pouco sensível a variações de memória, mas menos eficiente que o LR por executar uma passada extra sobre o arquivo invertido, realizando duas intercalações.

Entre futuros trabalhos na linha aqui investigada inclui-se a extensão dos algoritmos de indexação para tirar melhor proveito de máquinas multiprocessadas. Os algoritmos distribuídos projetados podem se beneficiar de mais de um processador por máquina escalonando as *threads* em processadores distintos, mas há melhores alternativas para o balanceamento da carga [11].

Ainda em relação a melhorias para os algoritmos distribuídos, podem-se procurar outras alternativas para a diminuição nos custos de sincronização do algoritmo RR, ou mesmo testá-lo em outros sistemas operacionais, de forma a se compararem os custos de invocação das primitivas de sincronização. Outros possíveis trabalhos ligados à manutenção do índice são a implementação da compressão descrita em [58] e o projeto de algoritmos que permitam a construção incremental de índices distribuídos. Por fim, um trabalho maior seria a criação de um sistema completo para a recuperação distribuída de informação utilizando os índices gerados e seguindo o modelo analítico proposto em [63].

Referências

[1] T. Anderson, D. Culler e D. Patterson. A case for NOW (Network Of Workstations). *IEEE Micro*, 15(1):54–64, Fevereiro de 1995.

[2] M. D. Araújo. IGREP – Um sistema de busca aproximada em textos indexados. Dissertação de Mestrado, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, Minas Gerais, Brasil, Setembro de 1997.

[3] M. D. Araújo, G. Navarro e N. Ziviani. Large text searching allowing errors. Em R. Baeza-Yates, editor, *IV South American Workshop on String Processing – WSP'97 – International Informatics Series*, volume 8, páginas 2–20, Valparaíso, Chile, Novembro de 1997. Carleton University Press.

[4] K. Arnold e J. Gosling. *The Java Programming Language*. Java Series from the Source. Addison-Wesley, Reading, Massachusetts, segunda edição, 1997.

[5] R. Baeza-Yates e G. Navarro. Block-addressing indices for approximate text retrieval. Em F. Golshani e K. Makki, editores, *Proceedings of the Sixth ACM International Conference on Information and Knowledge Management (CIKM'97)*, páginas 1–8, Las Vegas, USA, 1997.

[6] R. Baeza-Yates e B. Ribeiro-Neto, editores. *Modern Information Retrieval*. Addison-Wesley, Reading, Massachusetts, 1999.

[7] R. A. Barbosa. Desempenho de consultas em bibliotecas digitais distribuídas. Dissertação de Mestrado, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, Minas Gerais, Brasil, 1998.

[8] N. Belkin, P. Ingwersen e A. Pejtersen, editores. *Proceedings of the 15th ACM SIGIR International Conference on Research and Development in Information Retrieval*, Copenhagen, Dinamarca, Junho de 1992. ACM Press, New York.

[9] A. Bookstein. On the perils of merging boolean and weighted retrieval. *ASIS Journal*, 29(5):156–158, Maio de 1978.

[10] A. Bookstein. Fuzzy requests: An approach to weighting boolean searches. *ASIS Journal*, 31(7):240–247, 1980.

[11] E. Brown. Parallel and distributed IR. Em Baeza-Yates e Ribeiro-Neto [6], capítulo 9, páginas 229–256.

[12] D. R. Butenhof. *Programming with POSIX Threads*. Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1997.

[13] B. Cahoon e K. McKinley. Performance evaluation of a distributed architecture for information retrieval. Em Frei et al. [27].

[14] J. P. Callan, Z. Lu e W. B. Croft. Searching distributed collections with inference networks. Em E. Fox, P. Ingwersen e R. Fidel, editores, *Proceedings of the 18th ACM SIGIR International Conference on Research and Development in Information Retrieval*, páginas 21–28, Seattle, Washington, Julho de 1995. ACM Press, New York.

[15] Y. Choueka, A. S. Frankel, S. T. Klein e S. Segal. Improved hierarchical bit-vector compression of arithmetic coding. Em *Proc. 9th ACM SIGIR Conference on Information Retrieval*, páginas 88–97, Pisa, Italy, 1986. ACM Press, New York.

[16] T. H. Cormen, C. E. Leiserson e R. L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. McGraw-Hill, Cambridge, Massachusetts, 1990.

[17] W. B. Croft, A. Moffat, C. van Rijsbergen, R. Wilkinson e J. Zobel, editores. *Proceedings of the 21th ACM SIGIR International Conference on Research and Development in Information Retrieval*, Melbourne, Australia, Agosto de 1998. ACM Press, New York.

[18] W. B. Croft e C. van Rijsbergen, editores. *Proceedings of the 17th ACM SIGIR International Conference on Research and Development in Information Retrieval*, Dublin, Irlanda, Julho de 1994. ACM Press, New York.

[19] Z. J. Czech, G. Havas e B. S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182:1–143, 1997.

[20] Z. Czech, G. Havas e B. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43:257–264, Outubro de 1992.

[21] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21:194–203, 1975.

[22] C. Faloutsos. Signature files. Em Frakes e Baeza-Yates [26], capítulo 4, páginas 44–65.

[23] E. A. Fox, Q. F. Chen, A. Daoud e L. Heath. Order preserving minimal perfect hash functions and information retrieval. *ACM Transactions on Information Systems*, 9(2):281–308, 1991.

[24] E. A. Fox, L. Heath, Q. F. Chen e A. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, 1992.

[25] E. Fox, R. Akscyn, R. Furuta e J. Leggett. Digital libraries. *Communications of the ACM*, 38(4):22–28, Abril de 1995.

[26] W. B. Frakes e R. Baeza-Yates, editores. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[27] H.-P. Frei, D. K. Harman, P. Schauble e R. Wilkinson, editores. *Proceedings of the 19th ACM SIGIR International Conference on Research and Development in Information Retrieval*, Zurich, Suíça, Agosto de 1996. ACM Press, New York.

[28] E. Gamma, R. Helm, R. Johnson e J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1995.

[29] S. W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, IT-12(3):399–401, 1966.

[30] G. H. Gonnet, R. Baeza-Yates e T. Snider. New indices for text: PAT trees and PAT arrays. Em Frakes e Baeza-Yates [26], capítulo 5, páginas 66–82.

[31] D. R. Hanson. *C Interfaces and Implementations*. Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1997.

[32] D. K. Harman e G. Candela. Retrieving records from a gigabyte of texto on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41(8):581–589, 1990.

[33] D. K. Harman. Overview of the first text retrieval conference. Em D. K. Harman, editor, *Proceedings of the TREC Text Retrieval Conference*, páginas 1–20, Washington, DC, 1992. National Institute of Standards Special Publication 500-207.

[34] D. K. Harman. Relevance feedback revisited. Em Belkin et al. [8], páginas 1–10.

[35] D. K. Harman, E. A. Fox, R. Baeza-Yates e W. Lee. Inverted files. Em Frakes e Baeza-Yates [26], capítulo 3, páginas 28–43.

[36] G. Havas, B. Majewski, N. Wormald e Z. Czech. Graphs, hypergraphs and hashing. Em *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'93)*, número 790 em Lecture Notes in Computer Science, páginas 153–165. Springer-Verlag, Berlin, 1993.

[37] J. Heaps. *Information Retrieval – Computational and Theoretical Aspects*. Academic Press, New York, 1978.

[38] M. Hearst, F. Gey e R. Tong, editores. *Proceedings of the 22th ACM SIGIR International Conference on Research and Development in Information Retrieval*, Berkeley, California, Agosto de 1999. ACM Press, New York.

[39] D. A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of IRE*, 40(9):1098–1101, 1952.

[40] Y. Jing e W. B. Croft. An association thesaurus for information retrieval. Em *Proceedings of RIAO 94*, páginas 146–160, 1994.

[41] P. Kantor. The logic of weighted queries. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-11(12):816–821, Dezembro de 1981.

[42] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, Massachusetts, segunda edição, 1998.

[43] K. L. Kwok. A network approach to probabilist information retrieval. *ACM Transactions on Information Systems*, 13(3):324–353, Julho de 1995.

[44] Z. Lu. *Scalable Distributed Architectures for Information Retrieval*. Tese de Doutorado, Departament of Computer Science, University of Massachussets Amherst, Amherst, Massachussets, EUA, Maio de 1999.

[45] Z. Lu, K. McKinley e B. Cahoon. The hardware/software balancing act for information retrieval on symmetric multiprocessors. Relatório Técnico TR98-25, University of Massachussets, Amherst, MA, 1998.

[46] Z. Lu e K. S. McKinley. Partial replica selection based on relevance for information retrieval. Em Hearst et al. [38], páginas 105–112.

[47] B. Majewski, N. Wormald, G. Havas e Z. Czech. A family of perfect hashing methods. *Computer Journal*, 39:547–554, 1996.

[48] U. Manber e E. Myers. Suffix arrays: a new method for on-line string searches. Em *ACM-SIAM Symposium on Discrete Algorithms*, páginas 319–327, San Francisco, 1990.

[49] U. Manber e S. Wu. GLIMPSE: A tool to search through entire file systems. Relatório Técnico TR 93-34, The University of Arizona, Department of Computer Science, Outubro de 1993.

[50] D. A. Menascé, V. A. Almeida e L. W. Dowdy. *Capacity Planning and Performance Modeling - From Mainframes to Client-Server Systems*. Prentice Hall, Englewood Cliffs, 1999.

[51] A. Moffat e T. Bell. In-situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, Agosto de 1995.

[52] A. Moffat e J. Zobel. Parameterised compression for sparse bitmaps. Em Belkin et al. [8], páginas 274–285.

[53] A. Moffat e J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 1996.

[54] D. R. Morrison. PATRICIA – Practical Algorithm To Retrieve Information Coded In Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[55] G. Navarro, E. Moura, M. Neubert, N. Ziviani e R. Baeza-Yates. Adding compression to block addressing inverted indices. *Information Retrieval, Kluwer Academic Publishers*. To appear.

[56] J. Pearl. *Probabilistic Reasoning In Intelligent Systems: Networks of Plausible Inference*. Morgan Kauffmann Publishers, San Mateo, California, 1988.

[57] M. Persin. Document filtering for fast ranking. Em Croft e van Rijsbergen [18], páginas 339–348.

[58] M. Persin, J. Zobel e R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, 1996.

[59] M. J. Quinn. *Parallel Computing: Theory and Practice*. Series in Computer Science. McGraw-Hill, New York, segunda edição, 1994.

[60] V. V. Raghavan e S. K. M. Wong. A critical analysis of vector space model for information retrieval. *Journal of the American Society for Information Science*, 27(5):279–287, 1986.

[61] B. Ribeiro-Neto, J. Kitajima, G. Navarro, C. Sant'Ana e N. Ziviani. Parallel generation of inverted files for distributed text collections. Em *Proceedings of the XVIII International Conference of the Chilean Society of Computer Science (SCCC'98)*, páginas 149–157, Antofagasta, Chile, 1998.

[62] B. Ribeiro-Neto, E. Moura, M. Neubert e N. Ziviani. Efficient distributed algorithms to build inverted files. Em Hearst et al. [38], páginas 105–112.

[63] B. Ribeiro-Neto e R. A. Barbosa. Query performance for tightly coupled distributed digital libraries. Em *Proceedings of the Digital Libraries Conference '98*, páginas 182–190, Pittsburgh, Pasadena, 1998.

[64] S. Robertson. On the nature of fuzz: A diatribe. *ASIS Journal*, 29(11):304–307, Novembro de 1978.

[65] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy e W. Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[66] G. Salton. *Automatic Text Processing: The transformation, analysis and retrieval of information by computer*. Addison-Wesley, Reading, Massachusetts, 1989.

[67] G. Salton e M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, 1983.

[68] C. R. G. Sant'Ana. Construção de arquivos invertidos distribuídos. Dissertação de Mestrado, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, Minas Gerais, Brasil, 1998.

[69] R. Sedgewick. *Algorithms in C*. Series in Computer Science. Addison-Wesley, Reading, Massachusetts, 1990.

[70] Projeto SIAM/DCC/UFMG: Sistemas de Informação em Ambientes de Computação Móvel. Página *Web*: `http://www.dcc.ufmg.br/siam`. Concessão MCT/FINEP/-PRONEX número 76.97.1016.00.

[71] A. Silberschatz e P. B. Galvin. *Operating System Concepts*. Addison-Wesley, Reading, Massachusetts, quarta edição, 1994.

[72] C. Stanfill. Partitioned posting files: A parallel inverted file structure for information retrieval. Em *Proceedings of the 13th ACM SIGIR International Conference on Research and Development in Information Retrieval*, páginas 413–428, Bruxelas, Bélgica, 1990. ACM Press, New York.

[73] C. Stanfill. Parallel information retrieval algorithms. Em Frakes e Baeza-Yates [26], capítulo 18.

[74] C. Stanfill, R. Thau e D. Waltz. A parallel indexed algorithm for information retrieval. Em *Proceedings of the 12th ACM SIGIR International Conference on Research and Development in Information Retrieval*, páginas 88–97, Cambridge, EUA, Junho de 1989. ACM Press, New York.

[75] W. R. Stevens. *Advanced Programming in the UNIX Environment.* Professional Computing Series. Addison-Wesley, Reading, Massachusetts, 1992.

[76] W. R. Stevens. *UNIX Network Programming – Networking APIs: Sockets and XTI*, volume 1. Prentice-Hall, Upper Saddle River, New Jersey, segunda edição, 1998.

[77] W. R. Stevens. *UNIX Network Programming – Interprocess Communications*, volume 2. Prentice-Hall, Upper Saddle River, New Jersey, 1999.

[78] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, terceira edição, 1997.

[79] A. S. Tanenbaum. *Modern Operating System.* Prentice Hall, Upper Saddle River, New Jersey, 1992.

[80] A. Tomasic e H. García-Molina. Performance of inverted indices in shared-nothing distributed text document information retrival systems. Em *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, páginas 8–17, San Diego, EUA, 1993.

[81] A. Tomasic e H. García-Molina. Performance issues in distributed shared-nothing information systems. *Information Processing & Management*, 32(6):647–665, 1996.

[82] H. Turtle e W. Croft. Evaluation of an inference network-based retrieval model. *ACM Transactions on Information Systems*, 9(3):187–222, Julho de 1991.

[83] C. van Rijsbergen. *Information Retrieval.* Butterwords, 1979.

[84] A. N. Vo e A. Moffat. Compressed inverted files with reduced decoding overheads. Em Croft et al. [17], páginas 290–297.

[85] I. Witten, A. Moffat e T. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Series in Multimedia Information and Systems. Morgan Kaufmann, San Francisco, California, segunda edição, 1999.

[86] S. K. M. Wong, W. Ziarko, V. V. Raghavan e P. C. N. Wong. On modeling of information retrieval concepts in vector spaces. *ACM Transactions on Database Systems*, 12(2):299–321, Junho de 1987.

[87] S. Wu e U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, Outubro de 1992.

[88] J. Xu e W. B. Croft. Query expansion using local and global document analysis. Em Frei et al. [27], páginas 339–348.

[89] J. Xu e W. B. Croft. Cluster-based language models for distributed retrieval. Em Hearst et al. [38], páginas 254–261.

[90] G. Zipf. *Human Behaviour and the Principle of Least Effort.* Addison-Wesley, 1949.

[91] N. Ziviani e B. Ribeiro-Neto. Text operations. Em Baeza-Yates e Ribeiro-Neto [6], capítulo 7, páginas 163–190.