UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

HEURÍSTICAS PARA APRIMORAR O MÉTODO BMW E SUAS VARIANTES

Lídia Lizziane Serejo de Carvalho

Março de 2015

Manaus - AM

UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

HEURÍSTICAS PARA APRIMORAR O MÉTODO BMW E SUAS VARIANTES

Lídia Lizziane Serejo de Carvalho

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, Instituto de Computação - IComp, da Universidade Federal do Amazonas, como parte dos requisitos necessários à obtenção do título de Mestre em Informática.

Orientador: Edleno Silva de Moura

Março de 2015

Manaus - AM

Ficha Catalográfica

Ficha catalográfica elaborada automaticamente de acordo com os dados fornecidos pelo(a) autor(a).

C331h

Carvalho, Lídia Lizziane Serejo de
Heurísticas para aprimorar o método BMW e suas variantes / Lídia Lizziane Serejo de Carvalho. 2015
45 f.: il. color; 29 cm.

Orientador: Prof. Dr. Edleno Silva de Moura
Dissertação (Mestrado em Informática) - Universidade Federal do Amazonas.

1. Recuperação de Informação. 2. Processamento de Consultas. 3. Índices Invertidos. 4. Sistemas de Busca. I. Moura, Prof. Dr. Edleno Silva de II. Universidade Federal do Amazonas III. Título

HEURÍSTICAS PARA APRIMORAR O MÉTODO BMW E SUAS VARIANTES

Lídia Lizziane Serejo de Carvalho

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO PROGRAMA DE PÓS-GRADUAÇÃO DO INSTITUTO DE COMPUTAÇÃO DA UNIVERSIDADE FEDERAL DO AMAZONAS COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM INFORMÁTICA.

Aprovado por:

Prof. Edleno Silva de Moura, D.Sc.

Prof. Altigran Soares da Silva, D.Sc.

Prof. Ilmério Reis da Silva, D.Sc.

MARÇO DE 2015

MANAUS, AM – BRASIL

A Deus e em especial aos meus pais Carlos Alberto Carvalho e Glanair Carvalho que sempre me apoiaram em todas as etapas acadêmicas e incentivaram-me a nunca desistir.

Agradecimentos

A Deus, que me proporcionou mais esta conquista, por ter me capacitado e ajudado a concluir este mestrado.
Aos meus pais, por todo apoio e carinho, e ao meu esposo Ígor Martins pelo companheirismo e motivação durante os momentos difíceis.
A todos os professores que, de alguma forma, influenciaram na minha aprendizagem. Em especial ao professor Edleno Moura pelas orientações e incentivos durante todo processo.
Aos meus amigos e colegas, que compartilharam momentos de alegria e de dificuldades nestes dois anos.
Enfim, a todos aqueles que participaram direta ou indiretamente em mais esta etapa de vida.

"Confia no Deus eterno de todo o seu coração
e não se apóie no seu próprio entendimento.
Lembre-se de Deus em tudo o que fizer,
e ele lhe mostrará o caminho certo." (Provérbios 3: 5-6)

Resumo

Nos últimos anos, pesquisas relacionadas ao processamento de consultas em máquinas de busca têm sido realizadas com o objetivo de desenvolver métodos que reduzam o seu custo. Este trabalho visa propor modificações para melhorar o desempenho do algoritmo Block-Max WAND (BMW), um dos algoritmos mais eficientes propostos na literatura. O algoritmo BMW utiliza heurísticas para descartar documentos da resposta durante o processamento de consultas, o que torna sua execução extremamente veloz. Nesta dissertação, serão propostas e experimentadas modificações nas heurísticas de descarte de documentos e redução na quantidade de memória utilizada para processar consultas pelo algoritmo BMW e suas variantes, buscando-se assim ganhos de desempenho.

PALAVRAS-CHAVE: Recuperação de Informação, Processamento de Consultas, Índices Invertidos, Sistemas de Busca.

Abstract

Several research efforts have been conducted in the literature to develop methods to reduce the cost of query processing in search engines. This research aims to propose modifications to improve the performance of the block-Max WAND (BMW) algorithm, one of the most efficient algorithms proposed previously. The BMW algorithm uses heuristics to discard the documents entries at query processing, which makes it extremely fast. In this dissertation, we propose and evaluate additional heuristics to improve the perfomance of BMW and your variant BMW-CS in an attempt to both further reduces query processing times and the amount of memory required for processing queries.

KEY-WORDS: Information Retrieval, Query Processing, Inverted Indexes, Search Engines.

Sumário

Lista de Figuras x

Lista de Tabelas xii

Lista de Algoritmos xiii

1 Introdução 1

1.1 Organização do Trabalho 3

2 Fundamentação Teórica 4

2.1 Sistemas de Buscas 4

2.1.1 Índice Invertido 4

2.1.2 *Score* (ou Peso) de um Termo 5

2.1.3 Modelos em RI 6

2.2 Compressão e Descompressão de Índices 7

2.3 Métodos de Poda 9

3 Trabalhos Relacionados 13

3.1 WAND 14

3.2 Block-Max WAND (BMW) 14

3.3 BMW-CS 16

4 Estudo de Limiares Iniciais para Poda 18

4.1 Ambiente de Experimentação 20

4.2 Resultados 21

5 Blocos de Tamanho Variáveis 26

5.1 Experimentos e Resultados 30

5.1.1 Algoritmo BMW 30
5.1.2 Algoritmo BMW-CS 34

6 Conclusão **39**
6.1 Trabalhos Futuros 41

Referências Bibliográficas **42**

Lista de Figuras

3.1 Listas invertidas para os termos da consulta “amarelo verde azul". 16

4.1 Processamento de consultas com limiar ótimo. 19
4.2 Desempenho do algoritmo BMW em diferentes abordagens para limiares de poda iniciais: BMW original, BMW com décimo/milésimo maior *score* e BMW com o mínimo *score* das top-k respostas para blocos fixos de 64, 128 e 256 documentos. 22
4.3 Desempenho do algoritmo BMW-CS com as abordagens décimo/milésimo maior *score* com camadas de blocos fixos de 64, 128 e 256 documentos. 23
4.4 Desempenho e total de blocos fixos do algoritmo BMW-CS. 24
4.5 Desempenho do algoritmo BMW-CS para blocos fixos com a primeira camada formada por 40% do índice. 25

5.1 Algoritmo BMW - Organização da lista invertida (blocos fixos). 26
5.2 Exemplo de lista invertida com blocos variáveis. 29
5.3 Desempenho do algoritmo BMW com a utilização das abordagem de variação do tamanho de blocos (máximo 1024 documentos). 31
5.4 Desempenho do algoritmo BMW com a utilização das abordagem da variação do tamanho de blocos (máximo 128 e 256 documentos). 32
5.5 Consulta hipotética A B C. 33
5.6 Desempenho do algoritmo BMW-CS variando o número de blocos fixos nas duas camadas do índice. 35
5.7 Número total de entradas das *skiplists* para cada porcentagem da primeira camada - blocos fixos de 64, 128 e 256 documentos. 35

5.8 Desempenho do algoritmo BMW-CS variando o número de blocos nas duas camadas do índice. 37
5.9 Número total de entradas das skiplists para cada porcentagem da primeira camada - blocos variáveis. 37

Lista de Tabelas

4.1 Número total de entradas das skiplists para blocos fixos de 64, 128 e 256 documentos. 22
4.2 Média do tempo de processamento em milisegundos calculada a partir da variação de 18% da primeira camada para top-1000 respostas. 23

5.1 Desempenho do algoritmo BMW para blocos de tamanho fixo de 64, 128 e 256 documentos. A quantidade de blocos é apresentada com *. 28
5.2 Blocos Variáveis - Média do tempo de processamento em milisegundos calculada a partir da variação de 18% da primeira camada para top-1000 respostas. 36

Lista de Algoritmos

1 Pseudocódigo da geração de *skiplists* de blocos fixos. 27

2 Pseudocódigo da geração de *skiplists* de blocos variáveis. 38

Capítulo 1

Introdução

Máquinas de buscas são impulsionadas a recuperar informação de qualidade para seus usuários e, ao mesmo tempo, obter respostas de maneira rápida para cada consulta processada. Tal tarefa torna-se um desafio, pois a produção de resultados de qualidade em geral requer a adoção de modelos de recuperação de informação sofisticados, os quais tendem a aumentar os custos computacionais. Somam-se como fatores complicadores nesse cenário, o grande número de consultas normalmente processadas diariamente por máquinas de busca e o tamanho crescente de suas bases de dados, havendo hoje em dia sistemas de busca que lidam com bases contendo dezenas de bilhões de documentos. Por essas razões, a melhoria do desempenho de máquinas de busca tem sido foco de diversas pesquisas encontradas na literatura como, por exemplo, os trabalhos desenvolvidos por Ding e Suel [10] e Rossi *et al.* [16].

Sistemas de busca têm como principal objetivo procurar informação útil ou relevante dentro de uma coleção de documentos, dada a necessidade de informação do usuário. Essa necessidade de informação é descrita por meio de palavras-chaves ou termos de uma consulta. A partir dessas palavras-chaves, o sistema analisa a coleção de documentos retornando, ao final do processo, um conjunto de documentos ordenados de acordo com algum valor estimado de relevância. Segundo Baeza-Yates *et al.* [3], a noção de relevância tem um papel central na área de Recuperação da Informação (RI), pois a relevância é um julgamento pessoal que depende da tarefa a ser resolvida e do contexto.

As máquinas de busca em geral adotam uma estrutura de dados que permite encontrar documentos que contém um determinado termo presente na consulta de forma rápida e eficiente. A estrutura mais adotada é o arquivo invertido, que contém, para cada palavra

da coleção, uma lista invertida. A lista invertida de um termo armazena dados sobre todos os documentos onde o termo ocorre.

Processadores de Consulta são implementados visando minimizar custos em relação ao tempo de resposta e uso de memória. Seu principal objetivo é tomar uma consulta especificada pelo usuário e fornecer uma resposta ordenada, buscando retornar os melhores resultados. Para isso, os processadores utilizam modelos de Recuperação da Informação que produzem funções de *ranking*. Esses modelos, como o modelo vetorial [17] e o modelo BM25[15], computam um peso para cada documento dada uma consulta. Este peso é uma estimativa do grau de relevância de um documento em relação a uma consulta. Em geral os sistemas de busca computam e retornam apenas as *top-k* respostas com maior peso, onde k é o número de documentos retornados.

Ding e Suel [10] propuseram o algoritmo Block-Max WAND (BMW), um algoritmo para processamento de consultas que apresenta bons resultados tanto para consultas conjuntivas (operador *and*) como disjuntivas (operador *or*). Variantes do método BMW têm sido propostas e estudadas na literatura, incluindo o BMW-CS [16], o qual reduz o tempo de processamento de consultas quando comparado ao BMW, mas tem como desvantagem um aumento significativo na quantidade de memória necessária para processar consultas e o fato de não ter sido projetado para processar consultas conjuntivas.

Algoritmos como o BMW utilizam técnicas de descarte de documentos (técnicas de poda) que estão diretamente relacionadas às atuais heurísticas que visam a redução de custos de processamento. Neste algoritmo, as listas invertidas são divididas em blocos com tamanhos fixos, geralmente com 64 ou 128 documentos, tal que cada bloco seja descomprimido separadamente durante o processamento de consultas, acelerando ainda mais o processamento. O algoritmo mantém uma estrutura de *heap* de mínimo que armazena as *top-k* respostas que serão retornadas ao usuário. O menor peso entre todos os documentos presentes no *heap* é chamado de limiar de poda. Documentos que apresentam peso inferior ao limiar de poda são descartados. Inicialmente, o limiar apresenta valor igual a zero, porém ele é atualizado quando o *heap* está cheio, ou seja, quando apresenta k respostas.

Este trabalho tem como principal objetivo estudar e propor melhorias ao algoritmo BMW e seu variante BMW-CS com a finalidade de reduzir o tempo de processamento e quantidade de memória utilizada ao processar consultas.

Como principais resultados, obteve-se as seguintes contribuições: (i) mudanças no limiar de poda inicial na etapa de processamento da consulta e (ii) métodos para a criação de uma estrutura de blocos variáveis na etapa de indexação dos termos da coleção.

1.1 Organização do Trabalho

A presente dissertação está estruturada como segue. O Capítulo 2 apresenta definições de termos inerentes a sistemas de busca e processamentos de consultas que são essenciais para a compreensão deste trabalho. O Capítulo 3 expõe um resumo dos principais trabalhos relacionados presentes na literatura. Nos Capítulos 4 e 5, explicamos as estratégias propostas para a solução do problema apresentado, descrevendo os experimentos realizados e os resultados obtidos. E por fim, no Capítulo 6, discutimos as conclusões e direcionamentos para trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta algumas definições dentro do contexto de sistemas de busca textual que foram base para o desenvolvimento deste trabalho.

2.1 Sistemas de Buscas

Em sistemas de busca para a *web* o usuário pode expressar sua necessidade de informação por meio de consultas. Dada uma consulta, o sistema de busca tem como principal objetivo retornar os top-k resultados mais relevantes ao usuário, onde k representa o número de documentos a serem retornados. Tais sistemas possuem duas principais etapas: a indexação (criação do índice invertido) e o processamento da consulta.

2.1.1 Índice Invertido

Sistemas de busca lidam com grandes coleções de documentos que acarretam no aumento do custo de processamento de consultas. Com o intuito de reduzir estes custos, a maioria dos sistemas de buscas utiliza uma estrutura de dados conhecida como índice invertido.

O índice invertido é formado por todos os termos (ou palavras) distintos da coleção formando o vocabulário do índice, e para cada termo, é associada uma lista invertida. A lista invertida, representada da Equação 2.1, é composta por pares $(docID, Freq)$ onde *docID* é o identificador do documento e *Freq* é a frequência do termo no documento. Cada par indica o documento onde o termo t ocorre e a quantidade de vezes que o termo

t aparece no documento.

$$L_t = (docID_1, Freq_{docID_1,t})(docID_2, Freq_{docID_2,t})...(docID_k, Freq_{docID_k,t}) \quad (2.1)$$

De acordo com Ceri *et al.* [8], o índice é uma visão lógica onde documentos de uma coleção são representados por meio de um conjunto de termos da indexação ou palavras-chaves, ou seja, qualquer palavra contida no documento texto. Para os autores, o pressuposto da indexação é que a semântica de ambos, os documentos e a necessidade do usuário, pode ser adequadamente expressa através de conjuntos de termos de indexação. Naturalmente, este conceito pode ser visto como uma simplificação considerável do problema. Neste trabalho, as palavras-chaves foram extraídas diretamente a partir do texto do documento. Alguns sistemas de recuperação representam um documento por meio de um conjunto completo de palavras que aparecem nele; no entanto, em coleções muito grandes, o conjunto de termos representativos é reduzido.

Um índice invertido pode ser definido da seguinte forma: primeiro, um dicionário de termos (também chamado de vocabulário ou léxico), V, é criado para representar todas as ocorrências únicas dos termos na coleção de documentos C. A frequência em que aparece cada termo $t_i \in V$ em C também é armazenado. Então, para cada termo $t_i \in V$, chamado *posting*, uma lista L_i, a lista invertida, é criada contendo a referência para cada documento $d_j \in C$ onde t_i ocorre. Além disso, L_i deve conter a frequência de t_i dentro de d_j. O conjunto de *postings* juntamente com sua lista invertida é chamado de índice invertido [8].

2.1.2 *Score* (ou Peso) de um Termo

Nem todos os termos apresentam o mesmo grau de importância para descrever o conteúdo de um documento, e essa análise de decisão de qual termo é mais importante não é trivial. Por exemplo, dado um cenário que apresenta uma coleção com um grande número de documentos, pode-se afirmar que um termo que está presente em todos os documentos não é útil para definir quais documentos devem ser retornados para o usuário, porém um termo que aparece em apenas poucos documentos limita um grupo de documentos que podem ser de interesse ao usuário.

De acordo com Baeza *et al.* [3], diferentes termos de indexação têm diferentes graus

de importância para fins de descrição do conteúdo de documentos. Este efeito é capturado por meio da atribuição de pesos numéricos a cada termo da indexação de um documento.

No intuito de caracterizar a importância dos termos, o peso quantifica esta importância para descrever o conteúdo semântico do documento. Dado k_i o termo de indexação e d_j um documento, $w_{i,j} \geq 0$ representa o peso associado ao par (k_i, d_j). Para um termo de indexação k_i que não aparece em um documento, $w_{i,j} = 0$ [3].

2.1.3 Modelos em RI

Segundo Baeza *et al.* [3], modelos em RI compõem um processo que visa a produção de uma função de *ranking*, isto é, uma função que atribui *scores* a documentos relacionados a uma consulta. Esse processo é dividido em duas principais tarefas: a concepção de um arcabouço lógico para representar documentos e consultas (ex: conjuntos, vetores ou distribuições probabilísticas) e a definição de uma função de *ranking* que computa o grau de similaridade (grau de proximidade) de cada documento em relação à consulta. Existem três modelos clássicos em RI: o *Booleano*, vetorial e probabilístico.

O modelo *Booleano* é um simples modelo de recuperação baseado na teoria da computação e álgebra *Booleana*. Baeza *et al.* [3] consideram que os termos da indexação estão presentes ou não nos documentos. Uma consulta q é composta por termos de indexação ligados por três conectivos *Booleanos*: *not*, *and* e *or*. Basicamente, neste modelo, a similaridade é um, $sim(d_j, q) = 1$, quando o modelo *Booleano* prevê que o documento d_j seja relevante a consulta q. Caso contrário, a predição é que o documento não seja relevante, ou seja, $sim(d_j, q) = 0$. Ao contrário do modelo *Booleano*, o modelo vetorial [17] computa o grau de similaridade entre cada documento armazenado no sistema e a consulta do usuário utilizando uma atribuição de pesos não binária. Este modelo representa um documento d_j e uma consulta q por meio de vetores e quantifica o grau de similaridade pelo *cosseno* do ângulo entre os vetores $\vec{d_j}$ e $\vec{q}$.

Uma das fórmulas de similaridade mais populares na literatura é o modelo probabilístico BM25 (BM - *Best Match*) [15]. Este modelo é o resultado de uma série de experimentos e variações aplicadas a um clássico problema do modelo probabilístico. Todos os algoritmos utilizados neste trabalho adotaram este modelo para o cálculo de similaridade. A fórmula para o BM25 que define a similaridade entre um documento D e uma consulta

Q está descrita nas Equações 2.2 e 2.3.

$$BM25(D,Q) = \sum_{i=1}^{n} IDF(q_i) * \frac{f(q_i,D) * (K_1 + 1)}{f(q_i,D) + K_1 * (1 - b + b * \frac{|D|}{avg_doclen})} \tag{2.2}$$

Onde:

$$IDF(q_i) = log(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}) \tag{2.3}$$

A Equação 2.3 define o valor de IDF (frequência inversa do documento) para um termo q_i na coleção. Onde N representa o número de documentos da coleção e $n(q_i)$ representa o número de documentos da coleção onde o termo q_i ocorreu.

A Equação 2.2 define a função que atribui um *score* a um documento que é usada para gerar o *ranking*. b é uma constante com valores no intervalo $[0,1]$ sendo K_1 e b constantes empíricas. $f(q_i,D)$ representa a frequência do termo q_i no documento D, $\mid D \mid$ representa a norma do documento e avg_doclen representa a norma média dos documentos da coleção. Nos experimentos realizados durante o desenvolvimento deste trabalho foram definidos os parâmetros $b = 0.75$ e $K_1 = 2$, conforme Baeza *et al.* [3].

2.2 Compressão e Descompressão de Índices

Muitas técnicas de compressão de índice têm sido estudadas na literatura pois é o meio mais compacto de indexação para grandes bases de dados textuais. A desvantagem da compressão é a necessidade de descomprimir os índices da lista durante o processamento de consultas, o que pode aumentar o tempo de processamento [22].

De acordo com Scholer *et al.* [18], a compressão do índice tem três principais benefícios de desempenho: (i) um índice comprimido requer menos espaço de armazenamento, (ii) os dados comprimidos fazem um melhor uso de largura de banda dos meios de comunicação disponíveis; mais informações podem ser transferidas por segundo do que quando os dados estão descomprimidos, (iii) a compressão aumenta a probabilidade de que parte do índice necessária para avaliar uma consulta já esteja em *cache* na memória, portanto, podendo evitar acessos ao disco. Assim, a compressão dos índices pode reduzir custos em sistemas de recuperação de informação.

Os autores Yan *et al.* [24] propõem uma avaliação das técnicas de compressão relacionadas à ordenação dos ID‘s dos documentos da lista invertida, e apresentam algumas oti-

mizações que tentam melhorar técnicas de compressão e descompressão existentes dentro deste contexto. A maioria das técnicas substitui cada docID (exceto o primeiro da lista) pela diferença entre ele e o docID anterior, chamado *d-gap*, e depois codificam o *d-gap* utilizando algum algoritmo de compressão de inteiros. Os autores afirmam que esse tipo de codificação resulta em uma maior taxa de compressão, pois diminui o valor médio que deve ser comprimido.

Zhang *et al.* [25] também fazem comparações entre alguns algoritmos de compressão de listas invertidas, incluindo novos variantes de algoritmos existentes. Eles implementaram métodos diferentes de compressão, incluindo codificação Variable-Byte [18], Simple9 [1], Rice [27], PForDelta [13, 28] e uma extensão do Simple9, chamada Simple16.

O algoritmo BMW, *baseline* deste trabalho, originalmente utiliza o método PForDelta, porém os métodos de compressão de índices implementados inicialmente na versão do BMW neste trabalho foram as codificações *Elias Gamma* e *Elias Delta*.

Uma das técnicas mais recentes de compressão de dados em sistemas em RI é a codificação PForDelta (PFD). PForDelta foi criada especificamente para a capacidade superescalar das CPUs modernas, realizando uma descompressão super rápida e ao mesmo tempo alcançando um tamanho menor de dados comprimidos. Primeiramente, determina-se um valor b de tal forma que a maior parte dos valores a serem codificados (pelo menos 90%) sejam menores que 2^b, encaixando-se em um campo fixo de b bits cada. Os valores restantes, chamados exceções, são codificados separadamente. A codificação é realizada armazenando-se cada inteiro como uma entrada de b bits. Cada entrada é, em seguida, inserida em uma lista de $[b*k]$ bits, onde k é usualmente escolhida como um múltiplo de 32. Se aplicarmos PFD para blocos contendo algum valor múltiplo de 32, então a descompressão envolve extrair grupos de 32 valores de b bits e, finalmente formar o resultado por meio da decodificação de um número pequeno de exceções. Yan *et al.* [24] propõe algumas modificações, implementação utilizada no algoritmo BMW, com o objetivo de alcançar melhor *perfomance* em termos de tamanho e velocidade.

A codificação *Elias* [11] é um método não parametrizado de codificação de inteiros. Métodos não parametrizados representam inteiros usando um esquema de codificação fixa. A codificação *Elias* pode ser dividida em *Gamma* e *Delta* [23].

Na codificação *Elias Gamma* um inteiro x é representado em duas partes: um cabeçalho com $\lfloor \log_2 x \rfloor$ zeros, seguido de uma representação binária de x. Por exemplo, a

representação do inteiro 9 é a sequência 0001001, visto que $\lfloor \log_2 9 \rfloor = 3$ sendo o cabeçalho 000, seguido da representação 1001 do número 9 em binário [20]. A codificação *Elias Delta* é um pouco mais longa que a codificação *gamma* para pequenos números inteiros, mas para inteiros maiores tais como os IDs dos documentos em um índice grande, a situação é inversa. Para um inteiro x, o código *Delta* também representa um inteiro em duas partes: $1 + \lfloor \log_2 x \rfloor$ usando o código *Gamma*, seguido de uma representação binária de x sem o bit mais significativo[23].

2.3 Métodos de Poda

Métodos de podas são criados para reduzir custos computacionais. Estes podem ser classificados como métodos dinâmicos e estáticos [9]. Métodos de poda dinâmicos mantém o índice completo armazenado em disco e usam heurísticas para evitar leitura de informação desnecessária durante o processamento de consultas. Os métodos estáticos, tentam predizer, durante a criação do índice, as entradas que não serão úteis ao processamento da consulta.

Carmel *et al.* [7] propuseram um método de poda estática para reduzir o tamanho dos índices em sistemas de RI. A idéia principal é remover das listas invertidas as entradas cujos *scores* de relevância sejam baixos de tal forma que o efeito na acurácia do sistema também seja pequeno. Blanco e Barreiro [4] também desenvolveram técnicas de poda estática nos termos do índice invertido. Estas técnicas reduzem o tamanho do índice identificando termos comuns, chamados *stop words*. A diferença entre esse método e o de Carmel é que o termo inteiro é removido do índice no lugar da remoção de uma simples ocorrência.

Moura *et al.* [9] propõem uma variação do método Carmel que tem como objetivo predizer um conjunto de termos que ocorrem juntos em consultas e usam essa informação para preservar documentos comuns nas listas invertidas desses termos. Portanto, além das top entradas de cada uma das listas para cada termo, o método preserva também as entradas que estão no topo da lista de outros termos relacionados.

Nguyen [14] apresenta a poda baseada em termo e a poda baseada em documento. A poda baseada em termo considera cada lista invertida independentemente e remove entradas menos importantes, enquanto que a poda baseada em documento considera cada

documento independentemente e remove termos menos importantes de cada documento. As duas abordagens assumem que todos os elementos são igualmente importantes, e assim a poda do índice deve manter alguma informação sobre todos os elementos, sejam entradas ou documentos. O autor propõe um método de poda baseado em *postings* generalizando essas duas abordagens.

Zheng e Cox [26] exploram a estratégia de poda baseada em documento, ou seja, os documentos menos importantes são removidos da coleção. Este método decide se todos os *postings* que apontam para um documento específico devem permanecer no índice. Para saber quais documentos devem ser removidos, os autores descrevem três métodos e afirmam que apesar da estratégia de eliminar documentos ser contraintuitiva, os experimentos realizados mostram que em alguns cenários os algoritmos implementados são competitivos ou até melhores do que outros algoritmos de poda baseado em *postings*.

Anh e Moffat [2] estudaram uma aplicação de poda dinâmica onde as entradas são ordenadas pelo impacto associado a cada documento, reduzindo a quantidade de memória utilizada por acumuladores e dados transferidos do disco. O método de poda dinâmica apresentado pelos autores divide a lista invertida em blocos. O bloco que contém os mais altos impactos são manipulados pelo modo disjuntivo (OR). Durante o processamento de cada lista invertida, o processamento se altera para o modo conjuntivo (AND). Neste modo, um candidato que já tenha sido avaliado aumenta seu valor de *score*. Os autores introduzem o modo de processamento chamado REFINADO, onde as entradas atribuídas a este modo são processadas somente se elas se relacionam com algum documento que pertença ao conjunto que apresenta as maiores pontuações. No final do processo, os documentos restantes de cada lista invertida são ignorados e a resposta é retornada ao usuário.

Strohman e Croft [19] contribuem com um novo método de poda dinâmica que aumenta em 15% o *throughput* da consulta sobre o método proposto por Anh e Moffat [2], mantendo o mesmo *rank* de resultados na mesma ordem que seria retornado em uma avaliação não otimizada. Os autores também mostram que as avaliações feitas pelo algoritmo apresentado por Anh e Moffat, são sete vezes mais rápidas no sistema deles do que a velocidade citada no artigo de Anh e Moffat.

O conceito de poda foi introduzido por Buckley e Lewit [6] para o processamento Termo a Termo (TAAT) e Turtle e Flood [21] para o processamento Documento a Docu-

mento (DAAT) [2].

No processamento TAAT as listas invertidas são ordenadas pelo peso do termo em ordem decrescente. Durante o processamento, são armazenados *scores* parciais que são acumulados até que todas as listas invertidas sejam processadas. Nesta abordagem, termos inteiros da consulta podem ser descartados como resultado da poda [2].

As mais modernas máquinas de busca obtêm as top-k respostas usando a abordagem documento a documento. Esta abordagem ordena a lista invertida pelo ID do documento e percorre as listas de *postings* dos termos da consulta em paralelo. Uma de suas formas mais clássicas, avalia para cada docID uma expressão Booleana, e se esta for satisfeita, o *score* final do documento poderá ser computado. Esse algoritmo, *naive* DAAT, faz uma varredura e descomprime a lista inteira de *postings* para cada termo da consulta e computa todos os docIDs das listas que satisfazem a expressão Booleana. Tal método é ineficiente para grandes coleções devido aos grandes custos relacionados a descompressão da lista e custos para computar os *scores* dos documentos. Visto isso, alguns algoritmos mais recentes de processamento de consulta desenvolvem técnicas de descarte de documentos, ou mesmo fornecem métodos que pulam eficientemente etapas deste processamento.

Broder *et al.* [5] implementam um algoritmo de processamento de consultas chamado WAND baseado na estratégia DAAT. O algoritmo processa as listas invertidas em paralelo identificando candidatos ao conjunto resposta. Estes são identificados por meio de uma avaliação preliminar, considerando somente informações parciais da ocorrência dos termos e informações dependentes da consulta. Uma vez que um candidato é identificado, ele é totalmente avaliado e seu *score* total é computado. O algoritmo itera em paralelo sobre os *postings* dos termos da consulta, mas a natureza da avaliação preliminar é que ela possibilita pular rapidamente uma grande porção de *postings* das listas invertidas.

Ding e Suel [10] propuseram uma modificação no algoritmo desenvolvido por Broder *et al.* [5] chamada de *Block-Max* WAND (BMW). Os autores criam uma estrutura na lista invertida chamada de *skiplist*. Tal estrutura divide a lista invertida em blocos e para cada um deles armazena-se a informação do máximo impacto entre os documentos do bloco possibilitando mais saltos de grandes partes da lista. Esta estrutura é integrada ao algoritmo WAND e apresenta grandes ganhos de *performance*. Este algoritmo é considerado estado da arte e é utilizado como baseline deste trabalho.

Algumas variações do BMW são encontradas na literatura, entre elas, o algoritmo

BMW-CS. Este algoritmo, proposto por Rossi *et al.* [16] modifica o algoritmo BMW dividindo a lista invertida em 2 camadas. A primeira camada contém os maiores *scores* da lista e a segunda possui as entradas restantes. O algoritmo BMW-CS possui melhor *performance* em relação ao BMW porém não preserva as top-k respostas requeridas pelo usuário. Isso ocorre porque alguns documentos que deveriam compor a resposta não são avaliados devido a dinâmica do algoritmo.

Os algoritmos WAND, BMW e BMW-CS serão descritos com mais detalhes no Capítulo 3 deste trabalho.

Capítulo 3

Trabalhos Relacionados

Diversos pesquisadores têm proposto estratégias para melhorar o desempenho do processamento de consultas em sistemas de busca nos últimos anos. Os métodos de processamento de consultas encontrados na literatura podem ser divididos em duas classes principais: o que utilizam estratégia DAAT (*Document-At-A-Time*) e os que utilizam estratégia TAAT (*Term-At-A-Time*).

Nos métodos que utilizam estratégia DAAT, as listas invertidas associadas à consulta processada são percorridas em paralelo. Cada lista tem um ponteiro para indicar o atual par documento/frequência que ainda não foi processado e todos esses ponteiros são movidos em paralelo conforme a consulta vai sendo processada. A lista invertida para cada termo da consulta é ordenada pelo *id* do documento, e para cada lista é armazenado o *score* máximo de todos os documentos presentes na lista. A cada momento no processamento de consultas DAAT, há um ponteiro para o próximo documento a ser processado em cada lista invertida associada a consulta.

Nos métodos que utilizam a estratégia TAAT, as listas invertidas são ordenadas pelo peso do termo em ordem decrescente. As respostas da consulta são obtidas sequencialmente percorrendo uma lista invertida a cada momento. Sua maior desvantagem é que esta estratégia requer muita memória para armazenar parcialmente *scores* alcançados por cada documento quando a lista é processada.

3.1 WAND

Um dos trabalhos mais importantes apresentados nos últimos anos propõe um método eficiente de processamento de consultas conhecido como WAND (*Weak* AND ou *Weighted* AND) [5], o qual é capaz de processar consultas conjuntivas e disjuntivas. O método WAND utiliza a estratégia DAAT e mantém os top-k documentos com os maiores *scores* a cada etapa do processamento da consulta, onde k é o número de documentos solicitados em um sistema de buscas. Tais documentos são armazenados em uma estrutura de *heap* de mínimo. Com essa estrutura, obtém-se sempre, direto da raiz, o documento com o menor *score* que já foi adicionado nas top-k respostas. O menor *score* do *heap* é tomado como um limiar de poda (*threshold*) para acelerar o processamento da consulta. Um novo documento é avaliado e inserido no *heap* somente se ele tem um *score* maior que este limiar.

Durante o processamento de uma consulta sempre é estabelecido um documento pivô. Este documento usa a informação do *score* máximo de cada lista onde ele possa ocorrer. Se o *score* máximo de todas as ocorrências desse documento entre as listas for maior que o limiar atual, o documento terá seu *score* avaliado. Caso isso não ocorra o documento é descartado e um novo pivô é selecionado. O algoritmo WAND garante que ocorrências anteriores em cada lista já foram avaliadas, e que cada ponteiro de documento representa o menor id do documento na lista que ainda não foi processado.

3.2 Block-Max WAND (BMW)

O algoritmo BMW [10] foi criado tendo como base o WAND[5], porém propõe uma solução otimizada que abrange principalmente uma modificação na estrutura de índices. Essa mudança divide a lista invertida em blocos (*skiplists*), e para cada bloco são armazenados o máximo *score* (*Block-Max Index*) e o último documento daquele bloco. Esta divisão consiste apenas em criar uma estrutura que aponte para cada conjunto de documentos da lista delimitado em um tamanho fixo. O *score* máximo de cada bloco é usado para diminuir custos no processamento de consultas descartando blocos inteiros de documentos que não apresentam peso suficiente para serem inseridos no *ranking* de respostas.

A primeira fase do algoritmo consiste em encontrar um documento candidato (pivô) para ser inserido na resposta. Se o *score* máximo da lista e o *score* máximo do bloco ao

qual pertence o pivô, respectivamente, forem superiores ao limiar de poda, o bloco será percorrido e o *score* do documento poderá ser calculado. No entanto, se o *score* máximo do bloco não for superior ao limiar de poda, o bloco inteiro será descartado e um novo candidato será selecionado. O limiar de poda é dinamicamente atualizado sempre que um documento é selecionado para compor os top-k resultados, consequentemente o menor *score* das k respostas será o novo limiar e o documento anterior de menor limiar será descartado da resposta.

Para exemplificar, supondo um cenário como mostra a Figura 3.1, o limiar de poda indica que é necessário que um documento tenha um *score* maior que 3.2 para que faça parte do atual *ranking* de respostas. A figura retrata um momento do processamento onde os ids dos documentos 20, 40 e 90 associados a cada termo da consulta “amarelo verde azul" serão avaliados. A lista invertida do primeiro documento a ser processado tem um *score* máximo de valor 2.5, ou seja, nenhum documento nessa lista poderá ultrapassar o limiar de poda. No entanto, o documento 40, caso exista também na lista invertida do termo “amarelo", poderá ter um *score* de 4.0 (2.5 + 1.5 > 3.2), maior que o limiar. Já que existe essa possibilidade, o documento 40 será o novo pivô. O algoritmo então verifica, se o *score* máximo do bloco do atual pivô é superior ao limiar de poda. A possibilidade do documento 40 ocorrer na lista invertida do termo “amarelo" mais o *score* máximo do bloco que ele pertence na lista do termo “verde" (2.3 + 1.0 > 3.2) é superior ao limiar, logo o documento 40 fará parte do *heap* de respostas se a soma do *score* do documento 40 na primeira lista com sua soma da segunda lista também tiver valor superior. Nesse exemplo o *score* do documento em cada lista foi dado: (2.3 + 1.0), que ultrapassa o limiar de poda, portanto o documento 40 entra no *heap* de respostas e o documento que tinha *score* de 3.2 é descartado da resposta. Os ponteiros serão rearranjados para os próximos documentos após o pivô e novos documentos serão processados.

Os autores apresentam bons resultados comparados a outros algoritmos como WAND. Também implementam as mesmas estratégias para consultas conjuntivas desenvolvendo melhores resultados para consultas com um número menor de termos.

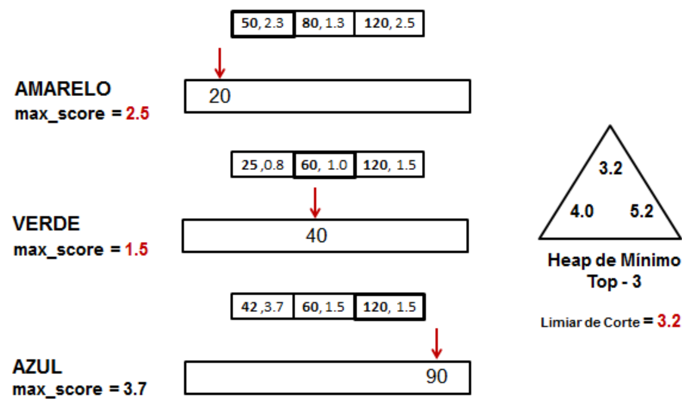


Figura 3.1: Listas invertidas para os termos da consulta “amarelo verde azul".

3.3 BMW-CS

Alguns autores exploram a abordagem do particionamento da lista invertida em mais de uma camada para acelerar o processamento de consultas. O algoritmo BMW_CS [16] apresenta, nesta linha de pesquisa, a organização da lista invertida em duas camadas, a primeira camada tem tamanho menor e é criada usando entradas que apresentam os maiores *scores* das listas invertidas provenientes de cada termo da consulta, enquanto a segunda camada têm um índice muito maior e contém as entradas restantes.

BMW_CS é baseado no algoritmo BMW com seleção de candidatos e suas duas camadas são disjuntas, ou seja, os documentos com maiores *scores* só estão presentes na primeira camada e os demais na segunda camada, e assim, como no BMW, os autores usam divisão por blocos nas duas camadas para acelerar ainda mais o processamento. Além disso, são armazenados o *score* máximo de cada lista invertida tanto para a primeira quanto para a segunda camada. O valor para a segunda camada é importante pois podem existir documentos que estão na primeira camada mas também ocorrem na segunda camada para outros termos da consulta.

Primeiramente o algoritmo compõe a primeira camada por meio de uma função que estima o tamanho da primeira camada para que a divisão das camadas assegure um custo menor em processamento. Logo na primeira fase, o algoritmo usa a primeira camada para selecionar candidatos que possivelmente farão parte dos top-k resultados. Verifica-se nessa fase, na primeira camada, se um documento a ser processado mais a probabilidade dele ocorrer nas demais listas da segunda camada é maior que o limiar de poda (menor *score* do *heap*). A primeira análise é realizada por meio do *score* máximo de cada lista,

onde o *score* máximo da primeira camada somado ao *score* máximo das demais listas na segunda camada deverá ser maior que o limiar de poda. Esta análise indica o possível documento a ser adicionado no *heap*, o pivô. Quando este é selecionado, um *score* máximo é computado, definido pelos autores como *upper_score*. Este valor é obtido por meio do *score* máximo do bloco do documento que está sendo processado na primeira camada mais o *score* máximo que um documento poderia ter para os outros termos da consulta na segunda camada. Sempre que o *upper_score* for maior que o limiar de poda, um novo documento será adicionado à lista de candidatos. Dado um documento pivô, calcula-se seu *score* completo por meio da soma de cada ocorrência desse documento nas listas invertidas na primeira camada. Se o *score* completo é maior que o limiar de poda, um novo documento é adicionado ao *heap* de respostas.

Ao final da primeira fase, com todos os documentos já processados, são excluídos os documentos na lista de candidatos que apresentam *upper_score* inferior ao limiar de poda. Este limiar apresenta o *score* mínimo do *heap* tendo em vista que os documentos inseridos eram da primeira camada, ou seja, documentos com pesos maiores.

Na segunda fase um novo *heap* é criado e a lista de candidatos é percorrida comparando cada documento com as listas invertidas dos termos da consulta na segunda camada. Soma-se para cada documento seu *score* com o *score* máximo do bloco em todas as listas invertidas em que o documento ocorre na segunda camada. Se a soma for superior ao limiar de poda o documento poderá ter seu *score* calculado e poderá ser adicionado ao *heap* de respostas.

Uma observação a ser feita na abordagem do BMW-CS é que apesar do bom desempenho apresentado pelos autores, ele não preserva todas as top-k respostas de uma consulta. Isso ocorre porque documentos que não apresentam *score* suficiente para fazer parte da primeira camada mas estão presentes na segunda camada não são analisados durante o processamento. Ou seja, podem existir documentos na segunda camada que, ao terem seus *scores* completos computados, poderiam ser inseridos a resposta.

Capítulo 4

Estudo de Limiares Iniciais para Poda

Neste capítulo, é descrita a abordagem para um novo limiar de poda inicial, visando o descarte de documentos no processamento de consultas. Esta é avaliada na Seção 4.2 por meio de experimentos de desempenho.

O limiar de poda possibilita o descarte de documentos que não apresentam *score* suficiente para serem inseridos nas top-k respostas para uma determinada consulta. O limiar de poda inicial no algoritmo BMW e BMW-CS possui valor igual a zero, o que faz com que haja custos e poucos descartes de documentos no inicio do processamento. Este é atualizado durante o processamento com o valor mínimo entre os k documentos de maior *score* dentre os já processados, k sendo o número de respostas que se deseja obter. Documentos avaliados que não apresentem valor superior ao limiar de poda são descartados. Na medida em que mais documentos são processados, o limiar de poda tende a subir até atingir o valor mínimo dentre os top-k resultados para a consulta a ser processada. A presença de um limiar inicial baixo faz com que menos documentos sejam descartados no inicio do processamento de consultas pelo algoritmo BMW e seus variantes.

A estratégia para adoção de limiar inicial foi aplicada em dois cenários: para top-10 e top-1000 documentos retornados durante o processamento de consultas. Para isso, foram capturados o k-ésimo maior *score*, $k = 10$ quando o algoritmo era executado no cenário de top-10 respostas, e $k = 1000$ para top-1000 respostas, de cada entrada presente na estrutura de índice invertido. Dada uma consulta, estima-se um valor inicial de *score* mínimo tomando-se o maior k-ésimo *score* dentre os termos da consulta. Tal escolha garante que o limiar inicial não irá descartar respostas, pois sabe-se que há pelo menos k entradas com score pelo menos igual ao limiar inicial escolhido. Por outro lado, a adoção

de tal limiar permite que se descarte a priori entradas que seriam levadas em consideração quando adotado um limiar inicial igual a zero.

Além de propor e experimentar estratégias para determinar um limiar inicial, fez-se também um estudo para verificar quanto se pode ganhar em desempenho com esse tipo de estratégia. Para isso, foi armazenado o mínimo *score* entre as k respostas obtidas no processamento de consultas e depois as mesmas consultas foram novamente processadas, porém com o limiar de poda inicial atualizado com os valores de mínimo *score* correspondentes para cada consulta. Denominamos este valor como limiar ótimo, por ser um indicativo do maior ganho que poderíamos obter com a utilização da estratégia de limiar de poda inicial. Como o limiar ótimo é apenas um valor de referencia, ele não pode ser usado como limiar inicial. Isto ocorre pois, para obtermos este limiar, executamos o algoritmo duas vezes. Na primeira vez, capturamos o menor *score* de cada resposta apresentada ao final do processamento, e posteriormente executamos novamente o algoritmo com o valor de limiar inicial atualizado. Portanto, o limiar inicial ótimo apenas orienta o quanto podemos melhorar no tempo de respostas com a inserção de um limiar inicial. A Figura 4.1 exemplifica esse método. A resposta com o limiar inicial ótimo é alcançada por meio de duas etapas: primeiramente, quando se deseja obter as top-3 respostas para uma consulta "A B C", o algoritmo inicia com um limiar igual a zero e, ao final do processo, o valor de menor impacto da resposta é armazenado. Na segunda etapa, novamente a consulta é processada, porém o valor armazenado é atribuído ao limiar inicial para a mesma consulta. Ao final do processo, obtemos a mesma resposta com um tempo de processamento menor.

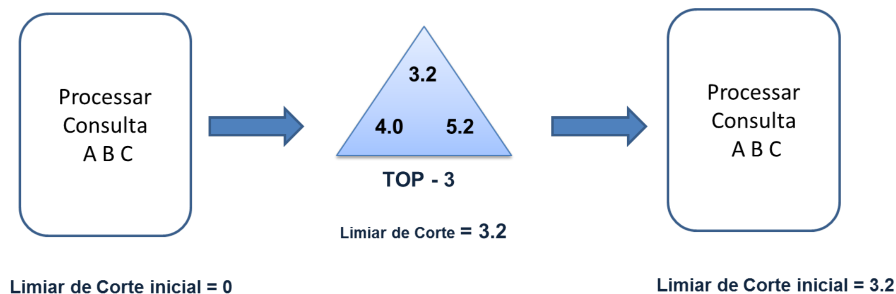


Figura 4.1: Processamento de consultas com limiar ótimo.

4.1 Ambiente de Experimentação

Nesta seção, apresentamos os experimentos realizados para avaliar o ganho do desempenho produzido pelas estratégias de poda estudadas. Como os *baselines* foram implementados e os autores apresentam resultados com a utilização de blocos fixos de 128 documentos, os experimentos foram desenvolvidos principalmente para este cenário. Para uma melhor análise dos dados, também estudamos a abordagem em cenários de blocos fixos com menos e mais documentos (64 e 256). O principal objetivo consistia em inferir se a introdução de limiares iniciais afetaria de forma positiva a performance dos algoritmos.

O trabalho foi desenvolvido com base no arcabouço de sistemas de busca já implementado no laboratório BDRI-UFAM com apenas algumas modificações no sistema. O sistema adotado processa consultas utilizando o algoritmo BMW ou ainda o variante BMW-CS. Utilizou-se nos experimentos a coleção de referência TREC GOV2. Esta é composta por 25.205.179 milhões de páginas coletadas do domínio .gov no início de 2004. A TREC GOV2 possui 426 GB de texto, compostos por páginas HTML e texto extraído de páginas no formato PDF e postscript (PS). O índice inteiro possui aproximadamente 7 GB de listas invertidas e o vocabulário é composto por cerca de 4 milhões de termos distintos. Nós aplicamos o algoritmo para redução de sufixos Porter Steammer Porter [2006], para reduzir o tamanho do vocabulário.

Foi selecionado aleatoriamente um conjunto de 10000 consultas extraídas da coleção TREC 2006 *efficiency queries*. Todos as *stopwords* dessas consultas foram retiradas. Durante o processamento de consultas, o índice inteiro é carregado para a memória. Isso evita qualquer interferência na avaliação do tempo de processamento de uma consulta. Todas as configurações foram escolhidas por serem similares às adotadas em trabalhos similares [19, 10], o que torna mais fácil a comparação entre os métodos estudados. Os experimentos foram executados em uma máquina de 24-cores Intel(R) Xeon(R), com processador X5680, 3.33GHz e 64GB de memória. Todas as consultas foram processadas em um único core.

Um dos parâmetros avaliados foi o tamanho das k respostas retornadas pelo sistema. Todos os experimentos foram executados em cenários que retornavam top-10 e top-1000 respostas. A recuperação das top-1000 respostas foi incluída para simular um ambiente onde o conjunto top-1000 é utilizado como entrada para um método de classificação mais sofisticado. O cenário de top-10 respostas foi incluído para simularmos um cenário mais

comum, onde o usuário está interessado apenas em uma pequena lista de resultados para a sua consulta. Os algoritmos foram avaliados em termos de tempo de resposta e custos de memória relacionados ao processamento de consultas.

As estratégias foram implementadas no algoritmo BMW, proposto por Ding e Suel[10], e o algoritmo BMW-CS, proposto por Rossi et.al. [16]. Estas foram desenvolvidas em C++ tendo como base trabalhos anteriores que desenvolveram o algoritmo BMW e BMW-CS, com resultados validados empiricamente e experimentos de desempenho. A versão criada para os dois algoritmos, que foi utilizada durante a execução dos experimentos, não apresenta técnicas de compressão e descompressão, mesmo assim possui tempo de processamento similar aos *baselines*. Assim como os algoritmos originais, mantemos a utilização do modelo probabilístico Okapi BM25 descrito na Seção 2.1.3, como função de similaridade. Para o algoritmo BMW-CS, variamos a primeira camada de 2 em 2 até 50% do índice completo nos experimentos.

4.2 Resultados

Nesta seção, são apresentados e discutidos os resultados dos experimentos realizados para a proposta de limiar de poda inicial.

As Figuras 4.2(a) e 4.2(b) mostram ganhos nas estratégias propostas em relação ao algoritmo BMW sem limiar (BMW Original) para as top-10 e top-1000 consultas, respectivamente. As figuras apresentam os experimentos realizados para blocos fixos de 64, 128 e 256 documentos. Blocos fixos com 64 documentos resultam em menores tempos de processamento de consultas, porém como indicado na Tabela 4.1, formam mais blocos no índice invertido, ou seja, requerem mais memória. De maneira oposta, blocos fixos com 256 documentos formam menos blocos, no entanto possuem tempo de processamento maior. Quanto maior a quantidade de blocos formados (número de entradas da estrutura *skiplist*), maior é o custo de memória usado no processamento de consultas, pois será necessário mais espaço para o armazenamento destes dados. Como já foi discutido no capítulo anterior, geralmente o algoritmo BMW é executado com blocos fixos de 128 documentos. Nesta configuração, observa-se que com um limiar inicial ótimo, podemos ter uma redução aproximada de 18% do tempo de processamento para as top-10 respostas e 16% para as top-1000 respostas. Como podemos observar também na Figura 4.2, em-

pregar a estratégia de décimo/milésimo para blocos de 128 entradas apenas teve impacto para top-1000 respostas, com uma redução aproximada de 5,5% em relação ao algoritmo original. Os maiores ganhos no uso desta estratégia estão relacionados a blocos fixos de tamanho 64, onde temos reduções aproximadas de 5,2% e 7,8% para top-10 e top-1000 respostas, respectivamente.

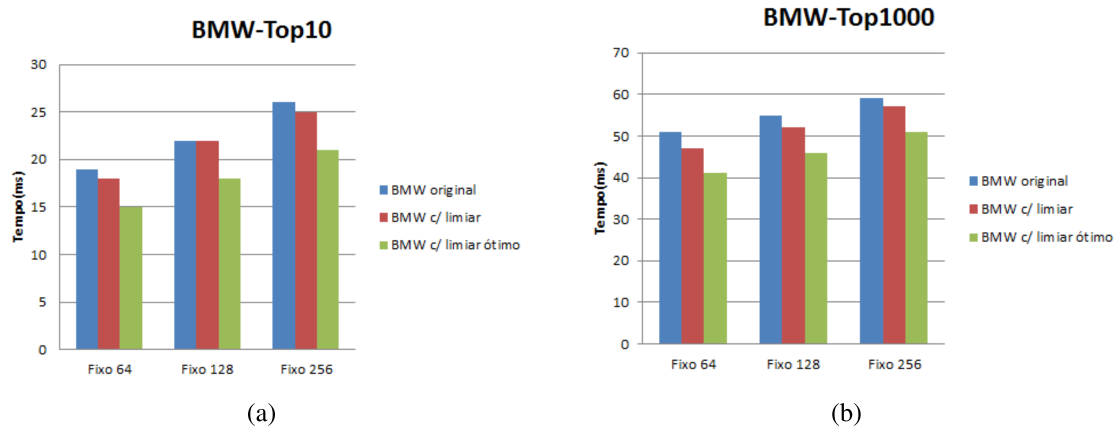


Figura 4.2: Desempenho do algoritmo BMW em diferentes abordagens para limiares de poda iniciais: BMW original, BMW com décimo/milésimo maior *score* e BMW com o mínimo *score* das top-k respostas para blocos fixos de 64, 128 e 256 documentos.

Fixo 64	Fixo 128	Fixo 256
23.595.719	11.798.404	5.899.760

Tabela 4.1: Número total de entradas das skiplists para blocos fixos de 64, 128 e 256 documentos.

Podemos observar na Figura 4.3 o impacto da inserção do milésimo e do décimo maior *score* para cada porcentagem de variação do tamanho da primeira camada do algoritmo BMW-CS. Para blocos fixos de 64, 128 e 256 documentos, nas duas camadas, os maiores ganhos do método de limiar inicial são obtidos para top-1000 respostas. Dentro deste cenário, para a análise de tempo de processamento, calculamos a média de tempo entre as variações de 18% e 50% da primeira camada para todas as opções de implementação, como mostra a Tabela 4.2. A tabela também revela que a introdução do milésimo como estratégia de limiar inicial reduz o tempo de processamento de consultas em até aproximadamente 17% .

Temos nas Figuras 4.4(a) e 4.4(b) respectivamente, o comportamento da implementação da estratégia de limiar inicial e o total de blocos formados para os três cenários, quando variarmos o tamanho da primeira camada. Percebe-se que não há muita oscilação

	Fixo 64	Fixo 128	Fixo 256
BMW-CS original	54,65	57,94	59,94
BMW-CS c/ limiar	45,12	47,82	49,88
Redução (%)	17,44	17,46	16,78

Tabela 4.2: Média do tempo de processamento em milisegundos calculada a partir da variação de 18% da primeira camada para top-1000 respostas.

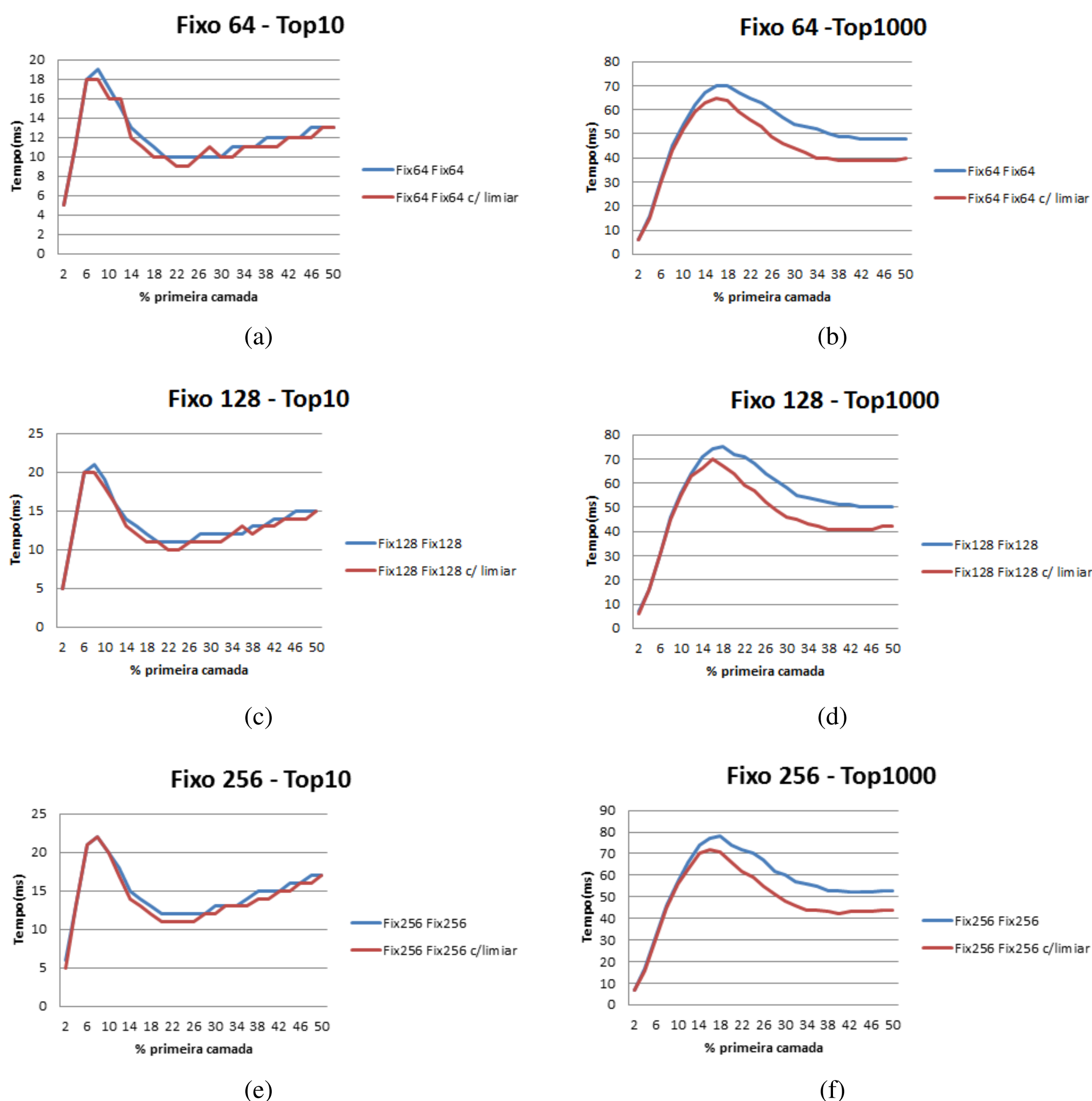
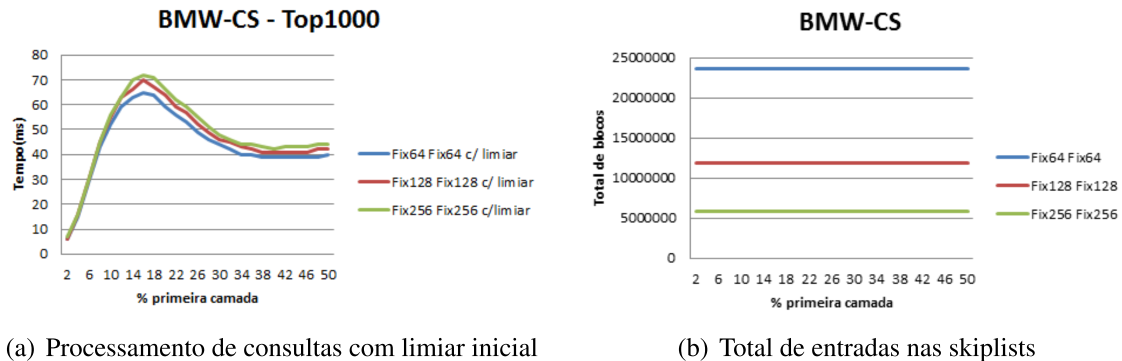


Figura 4.3: Desempenho do algoritmo BMW-CS com as abordagens décimo/milésimo maior *score* com camadas de blocos fixos de 64, 128 e 256 documentos.

no total de blocos na medida em que aumentamos a porcentagem da primeira camada. Nota-se ainda que o uso de blocos fixos de 64 entradas gera custos de memória expressivos e dada sua semelhança em relação ao tempo de processamento com blocos fixos de 128 e 256 (Figura 4.4(a)), ele não se mostra como uma boa opção de implementação. Tal cenário, apresenta a média aproximada de 23,6 mil blocos, enquanto blocos de 128 e

256 documentos formam aproximadamente 12 mil e 6 mil blocos, respectivamente. Isto indica que blocos fixos de 256 reduzem em até aproximadamente 50% o total de blocos do *baseline* com limiar, o qual implementa blocos fixos de 128 entradas. Portanto, como verifica-se na Figura 4.4(a), visto que o tempo de processamento entre os cenários são bem similares, podemos inferir que a melhor implementação é aquela que apresenta o menor número de blocos criados, ou seja, o cenário de blocos fixos com 256 entradas.



(a) Processamento de consultas com limiar inicial

(b) Total de entradas nas skiplists

Figura 4.4: Desempenho e total de blocos fixos do algoritmo BMW-CS.

Como foi discutido, o ganho de tempo de processamento em relação a introdução do limiar para o algoritmo BMW-CS foi melhor visualizado para top-1000 respostas. Para entendermos melhor este comportamento, temos na Figura 4.5 o experimento realizado no algoritmo BMW-CS com blocos fixos de 128 entradas, com a primeira camada formada por 40% do índice. Excepcionalmente, este experimento envolve outros cenários além de top-10 e top-1000 respostas retornadas no processamento de consultas. Para o mesmo foram verificados o tempo de processamento para top-10, top-100, top-1000, top-5000 e top-10000 documentos. Por meio do gráfico podemos inferir que quanto maior é a quantidade de resultados retornados (topo) para cada consulta, maior é o impacto da inserção do limiar inicial. O limiar inicial nos processadores de consultas, como já foi discutido, só é atualizado após o *heap* de respostas estar completamente preenchido. A estratégia de limiar inicial produz resultados melhores para um topo maior, pois quando o topo é pequeno, por exemplo *heap* que mantém 10 respostas, teremos poucos descartes até que o *heap* fique cheio. No entanto, quando deseja-se obter as top-5000 respostas, por exemplo, o *heap* demorará muito mais a ser preenchido e com isso o limiar inicial tendo um *score* de valor relevante (nesse caso, o cinco milésimo) trará mais descartes acelerando o processo de avaliações até que o *heap* tenha 5000 documentos e o limiar seja atualizado.

Finalmente, mesmo após o preenchimento do *heap*, quanto maior o número de respostas computadas, maior o número de entradas processadas até que o limiar de poda atinja o valor do maior k-ésimo *score* dentre os termos da consulta.

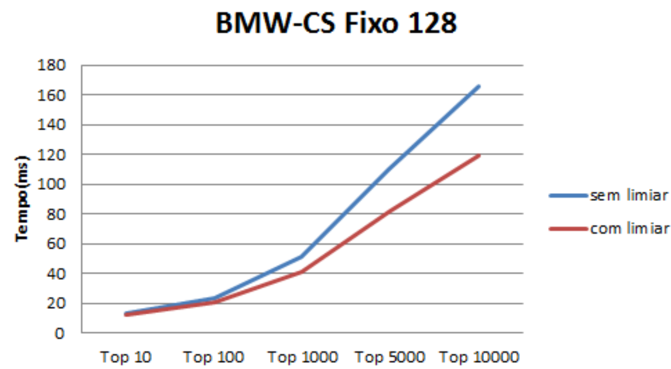


Figura 4.5: Desempenho do algoritmo BMW-CS para blocos fixos com a primeira camada formada por 40% do índice.

Capítulo 5

Blocos de Tamanho Variáveis

Neste capítulo, apresentamos as estratégias de variação do tamanho de blocos aplicadas ao algoritmo BMW e seu variante BMW-CS para o estudo dos impactos em relação ao tempo de processamento e ao uso de memória. Na Seção 5.1, mostramos os experimentos desenvolvidos para avaliar o desempenho das estratégias propostas.

Como já foi mencionado, o algoritmo BMW apresenta uma estrutura chamada *skiplist*, a qual divide a lista invertida em blocos de tamanhos fixos, geralmente compostos por 128 documentos. Além de armazenar a informação do *score* máximo presente em cada termo da coleção, o algoritmo mantém a informação do último documento, o máximo *score* de cada bloco e variáveis que são utilizadas na compressão e descompressão de documentos (*offset* e *bitoffset*) como mostra a Figura 5.1.

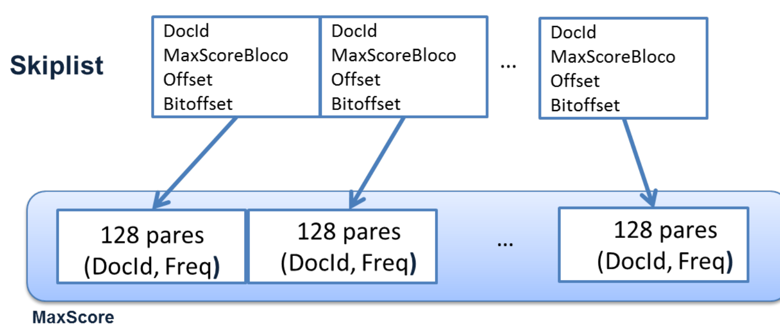


Figura 5.1: Algoritmo BMW - Organização da lista invertida (blocos fixos).

O Algoritmo 1 apresenta uma visão geral da implementação da estrutura de *skiplists* do algoritmo BMW e BMW-CS. Este cria uma estrutura de *skiplist* para cada termo da coleção (linha 2). As linhas 5 e 9 descrevem a captura e manipulação do primeiro documento de uma lista. Enquanto houver documentos em uma lista (linha 11), calcula-se o

score para cada documento, e uma variável é mantida para o armazenamento do máximo *score* de todos os documentos já analisados até o momento. Isto ocorre até que sejam lidos 128 documentos, etapa em que um bloco é criado (linha 19), ou seja, uma nova entrada é inserida na estrutura *skiplist*. Durante todas as etapas de criação dos blocos, podem ser manipuladas variáveis para a compressão de todos os documentos que estão sendo inseridos nos blocos. No final do processo, cada lista invertida apresenta uma *skiplist*, onde cada posição indica o término de cada bloco, com variáveis associadas para a posterior descompressão no processamento de consultas (linha 27). Como foi descrito na Seção 4.1, neste trabalho não utilizamos técnicas de compressão e descompressão de dados.

Algoritmo 1 Pseudocódigo da geração de *skiplists* de blocos fixos.

```
1:  Let L be the index
2:  for each 0 ≤ i ≤ |L| do
3:      sizeBlock ← 128;
4:      numDoc ← 0;
5:      load(lista[i]);
6:      if getDoc(lista[i], doc) then
7:          numDoc++;
8:      end if
9:      score ← BM25(lista[i],doc);
10:     maxScoreBlock ← score;
11:     while getDoc(lista[i],doc) do
12:         score ← BM25(lista[i],doc);
13:         numDoc++;
14:         if (maxScoreBlock < score) then
15:             maxScoreBlock ← score;
16:         end if
17:         //Creating of fixed blocks
18:         if (numDoc % sizeBlock = 0) then
19:             updateSkip(bufferSkip, doc, maxScoreBlock);
20:             maxScoreBlock ← 0;
21:         end if
22:     end while
23:     //Creating of fixed blocks
24:     if (numDoc % salto == 0) then
25:         updateSkip(bufferSkip, doc, maxScoreBlock);
26:     end if
27:     createSkip(bufferSkip, lista[i]);
28: end for
```

Ding e Suel [10] executam o algoritmo BMW, como já foi discutido, geralmente com blocos fixos de 128 documentos. Esta escolha é feita porque os autores avaliam tempo

de processamento e custos relacionados à memória. Podemos analisar na Tabela 5.1, que blocos fixos formados com menos documentos apresentam menores custos de processamento, porém formam um índice com mais blocos, e da mesma forma, blocos fixos de tamanho maior apresentam um aumento nos custos de processamento, no entanto o total de blocos criados é menor. Um dos fatores relacionados a custos de processamento é a escolha do documento pivô durante a execução dos algoritmos BMW e BMW-CS. Tal documento pode ou não ser inserido no *heap* de respostas, dependendo das avaliações realizadas como explicado na Seção 3.2. Estas avaliações são feitas a cada escolha de um novo pivô, acarretando em custos de processamento na medida em que estes são selecionados. Na última avaliação feita (verificar se o *score* do documento é superior ao limiar de corte), o documento pivô tem seu *score* total calculado antes de ser inserido no *heap* de respostas. Contudo, o fator agravante, que tem impacto principalmente no tempo de processamento, é que o bloco inteiro de 128 documentos deve ser percorrido para que o pivô seja encontrado e tenha seu *score* computado.

Alguns blocos são totalmente descartados quando o máximo *score* de um bloco não supera o limiar de corte durante o processamento de consultas. Com isso, percebe-se que blocos que apresentam máximo *score* do bloco baixo apresentam maior probabilidade de descarte. Neste caso, inferimos que blocos que apresentam *score* baixo adjacentes poderiam ser integrados para que, durante o processamento de consultas, mais documentos possam ser descartados de uma só vez reduzindo custos relacionados a tempo.

		Pivôs	**Tempo(s)**
Blocos de 64	Top-10	369.781.344	20
23.595.719*	Top-1000	1.002.528.576	51
Blocos de 128	Top-10	426.335.552	22
11.798.404*	Top-1000	1.116.841.472	55
Blocos de 256	Top-10	495.183.424	27
5.899.760*	Top-1000	1.223.783.040	59

Tabela 5.1: Desempenho do algoritmo BMW para blocos de tamanho fixo de 64, 128 e 256 documentos. A quantidade de blocos é apresentada com *.

A idéia principal para a variação dos blocos está associada a relação do máximo *score* de cada bloco. Blocos adjacentes que apresentam máximo *score* do bloco baixos são agrupados para se obter mais descartes no processamento das consultas. Para determinar quando um bloco tem *score* baixo ou não, utilizou-se o milésimo *score* de cada termo

da coleção como referência. Blocos que apresentam máximo *score* do bloco menor que o milésimo maior *score*, entre todos os documentos da lista, são considerados blocos de baixo impacto, e portanto podem se unir a outro bloco.

A Figura 5.2 ilustra uma lista invertida para um termo A que está dividida em n blocos, cada um composto por 128 documentos. No exemplo, o milésimo maior *score* desta lista já foi computado e tem valor igual a 12. Neste cenário, para variar blocos, uni-se blocos que apresentam *score* abaixo do milésimo, ou seja, os blocos A_2 e A_3, visualizados no exemplo. Os blocos A_1, A_4 e An permanecem como estavam. O novo bloco $A_2 + A_3$ formado, terá 256 documentos e máximo *score* do bloco de valor igual a 8.

Figura 5.2: Exemplo de lista invertida com blocos variáveis.

Podemos observar no Algoritmo 2 o pseudo-código da proposta de variação do tamanho de blocos para o algoritmo BMW e BMW-CS. Este algoritmo é bem similar ao que já foi apresentado para o *baseline* (Algoritmo 1). Temos que as linhas 21 a 43 do Algoritmo 2 mostram a base para a implementação de blocos variáveis. Os tamanhos mínimo e máximo para os blocos devem ser definidos, neste exemplo, blocos de 128 e 1024 documentos, respectivamente, linhas 4 e 31. A cada bloco de 128 documentos (linha 21) o algoritmo faz as seguintes verificações: se o bloco tem *score* alto, se existem blocos adjacentes que apresentam *score* baixo para serem agrupados, ou se já foram lidos 1024 documentos. Durante essas verificações, é mantida a variável *previousFlag* indicando se existe ou não um bloco lido anteriormente que ainda não foi criado. Se o *score* máximo do bloco (*maxScoreBlock*) é superior ao milésimo maior *score* de todas as entradas, forma-se um bloco de 128 documentos (linha 22). Se algum bloco anterior ao atual tinha *score* baixo, ele também é formado. Um bloco sempre é formado quando 1024 documentos (linha 31) são lidos e todas as avaliações anteriores consideravam cada 128 documentos com *score* baixo.

O algoritmo BMW-CS apresenta a mesma estrutura de *skiplists*, no entanto esta é aplicada para as duas camadas, ou seja, existem duas estruturas de *skiplists* para cada lista invertida. Nele, a variação de blocos foi aplicada tanto a uma como as duas camadas,

observando-se o número de blocos criados durante a etapa de indexação e tempo de processamento de consultas. No estudo feito buscamos diminuir o número de blocos criados e, ao mesmo tempo, diminuir ou manter o tempo de processamento original.

5.1 Experimentos e Resultados

Nesta seção, são apresentados e discutidos os resultados dos experimentos realizados para a proposta de blocos variáveis. A Seção 4.1 discorre sobre dados referentes aos experimentos, como informações sobre a coleção e os *baselines* utilizados. Todos os experimentos realizados para a proposta de variação de blocos empregaram a estratégia de limiar inicial descrita no Capítulo 4. Os algoritmos BMW Original e BMW-CS Original utilizados como comparativos em relação as novas abordagens são os *baselines* estruturados com blocos fixos com 128 entradas.

Para o algoritmo BMW-CS, também apresentaremos um estudo sobre a utilização de blocos fixos nas suas duas camadas em cenários com menos e mais documentos do que os blocos de 128 documentos implementados nos *baselines*. Utilizamos a estratégia de blocos variáveis para as duas camadas, assim como variamos a primeira camada mantendo a segunda com blocos fixos, e vice versa.

5.1.1 Algoritmo BMW

Como primeira estratégia, estabelecemos um tamanho mínimo e máximo para cada opção de implementação da variação do tamanho dos blocos. Seguindo esta proposta, variamos blocos com no mínimo 16, 32, 64, 128, 256 e 512 documentos compondo ao máximo o total de 1024 documentos. A Figura 5.3 mostra um comparativo desta estratégia com o algoritmo BMW Original. Percebe-se o aumento no tempo de processamento para top-10 e top-1000 respostas. Para top-1000 respostas retornadas, por exemplo, ao aplicarmos a variação com blocos de 512 até 1024 documentos, o tempo aumenta de 52 milisegundos para um pouco mais de 60 milisegundos. Nota-se ainda na figura que, apesar do tempo de processamento aumentar, a utilização da estratégia diminui em até aproximadamente 80% o número total de blocos formados (Figura 5.3(c)) quando comparamos com o algoritmo BMW Original.

Com o intuito de tentar diminuir o tempo de processamento, desenvolveu-se a estra-

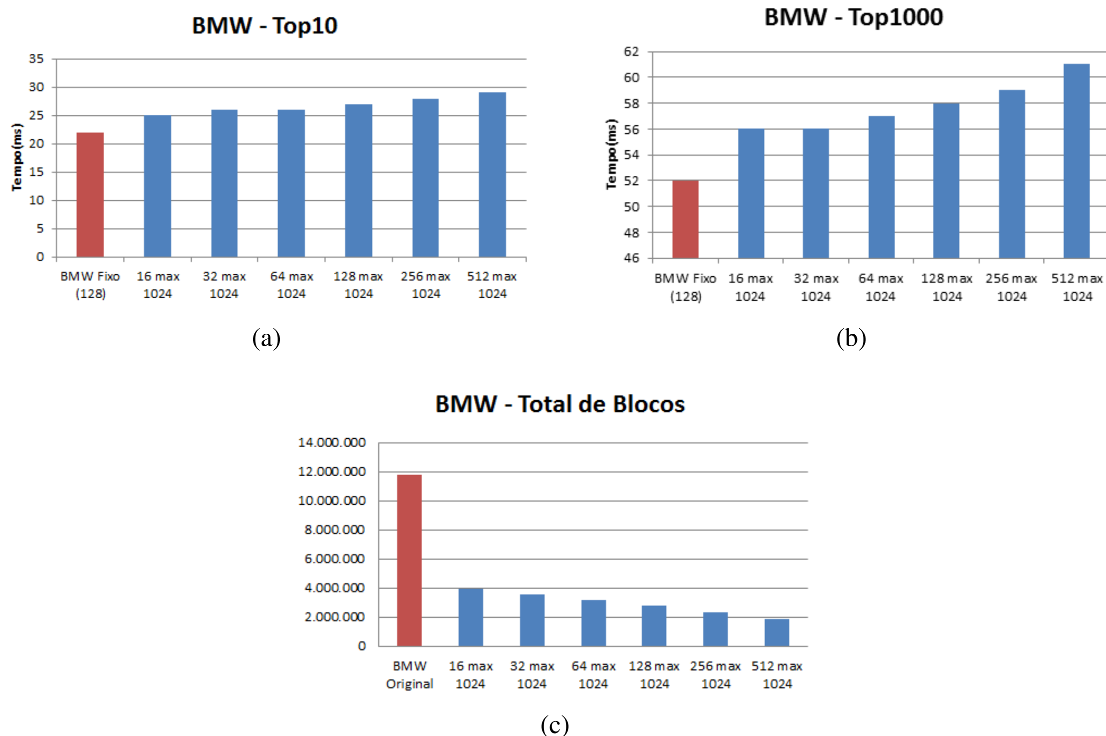


Figura 5.3: Desempenho do algoritmo BMW com a utilização das abordagem de variação do tamanho de blocos (máximo 1024 documentos).

tégia de variar o tamanho dos blocos mantendo-se um tamanho máximo de 128 e 256 documentos. Por exemplo, no caso do máximo ser 128 documentos, os blocos foram formados com 16, 32, 64 ou 128 documentos. Utilizamos esta estratégia partindo da ideia de que poderíamos reduzir, ou pelo menos manter, o tempo de processamento visto nas implementações de blocos fixos com os mesmos tamanhos.

Cada variação de blocos apresentada na Figura 5.4 indica que este método tem maiores reduções de tempo de processamento para blocos formados com o máximo 128 documentos para top-10 e top-1000 respostas (barras representadas com a cor azul). Ainda na figura, percebe-se que podemos manter um tempo de processamento similar ao BMW Original para blocos com o máximo de 256 documentos. Nota-se na Figura 5.4(c) que com esta opção de implementação reduzimos em até aproximadamente 44% o total de blocos formados. Por exemplo, a opção 32 max 256, que forma blocos de tamanho mínimo 32 e máximo 256 documentos, resultou em tempo de processamento igual ao obtido pelo BMW original (52 milisegundos). Contudo, o número de blocos gerados por tal opção foi 35% menor, gerando 7.676.292 blocos contra 11.798.404 do BMW original.

Analisamos algumas consultas com o intuito de identificar quais blocos eram aces-

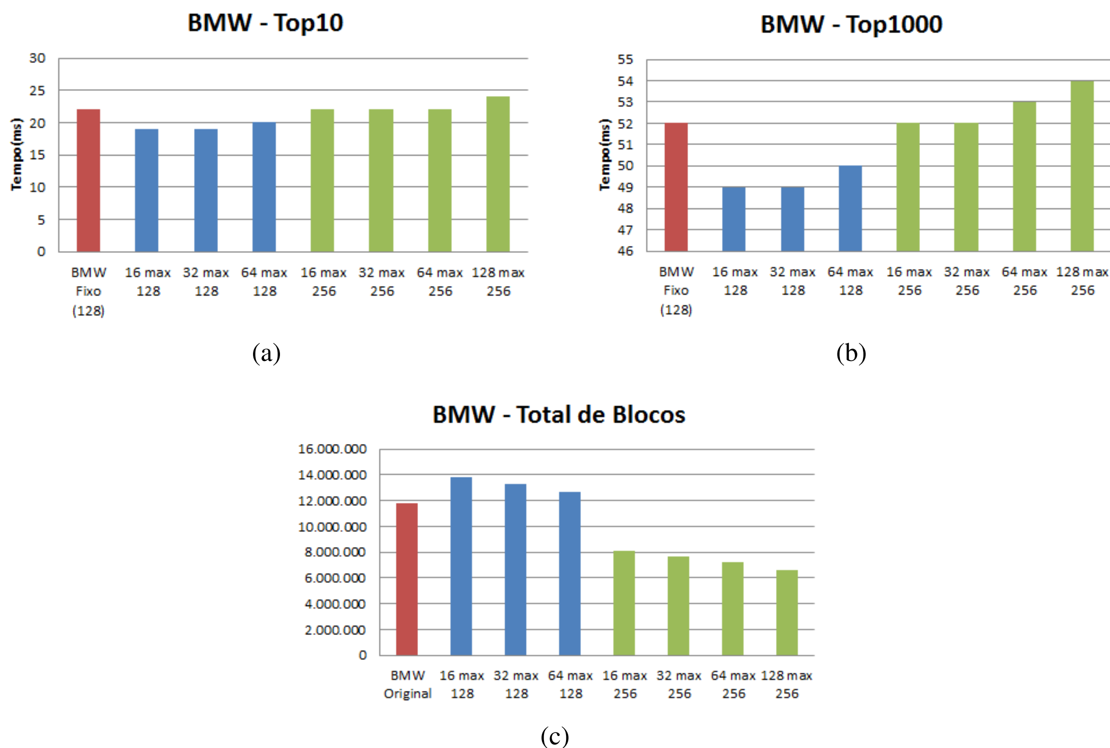
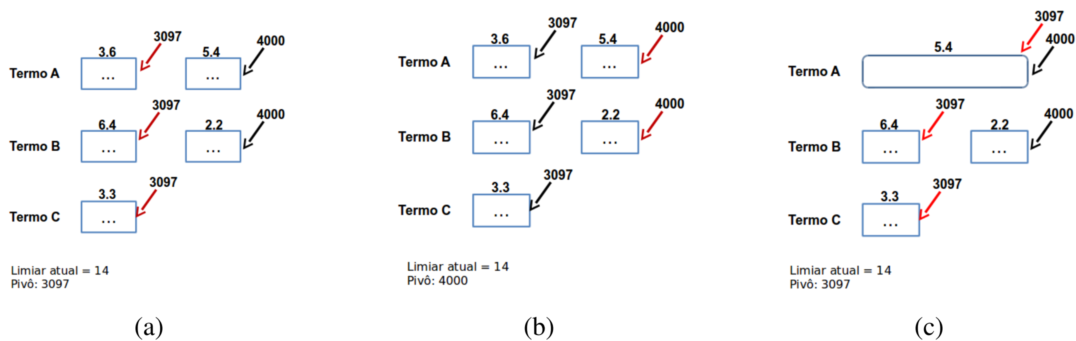


Figura 5.4: Desempenho do algoritmo BMW com a utilização das abordagem da variação do tamanho de blocos (máximo 128 e 256 documentos).

sados durante a etapa de processamento de consultas, e assim criar um índice com uma estrutura onde blocos adjacentes, não acessados, eram agregados em um só bloco. O objetivo foi manipular consultas para criar uma estrutura de blocos onde saberíamos a *priori* quais deles seriam descartados durante as avaliações. Ao final, teríamos um indicativo do tempo no qual uma consulta seria processada ao aplicarmos a estratégia de variar o tamanho de blocos.

A Figura 5.5 retrata um cenário hipotético do processamento de uma consulta "A B C" com blocos fixos de 128 documentos. No primeiro momento (Figura 5.5(a)), o menor documento do heap de respostas apresenta score igual a 14 e o pivô selecionado é o documento 3097. O documento 3097 está contido nas listas invertidas dos termos *A*, *B* e *C*, e temos o valor do score de cada bloco ao qual ele pertence. Como a soma do máximo *score* do bloco (3.6 + 6.4 + 3.3) de todas as ocorrências do documento pivô não é superior ao mínimo do heap (limiar atual), um novo pivô será selecionado. Quando o novo pivô é selecionado (documento 4000), a soma do máximo *score* de bloco de todas as ocorrências desse documento no índice também não supera o limiar atual. O algoritmo então segue buscando possíveis candidatos a serem retornados como resposta.

Neste cenário hipotético, os blocos que apresentam o documento 3097 e 4000 para o termo A não foram lidos, porém os blocos restantes foram visitados e portanto são marcados como lidos. Para a manipulação desta consulta, unimos os blocos que não foram lidos, pois intuitivamente pensamos que o algoritmo poderá pular mais documentos diminuindo custos de processamento. A Figura 5.5(c) representa esse novo cenário onde a consulta "A B C" apresenta uma estrutura com blocos variáveis. Neste momento do processamento da consulta, o documento pivô 3097 será avaliado. Na avaliação da soma dos máximos *scores* de bloco, agora com a estrutura de blocos alterada, temos as o resultado $5,4+6,4+3,3 = 15,1$ do documento em todos os termos da consulta apresenta valor superior ao limiar atual. Isso indica que o documento pivô terá seu *score* computado, ou seja, o algoritmo varrerá um bloco maior à procura do documento pivô, o que afetará seu desempenho.



(a) (b) (c)

Figura 5.5: Consulta hipotética A B C.

O exemplo da Figura 5.5 mostra que alterações no tamanho dos blocos podem gerar efeitos colaterais no desempenho dos algoritmos, dado que elas também alteram os valores de score máximo dos blocos e, consequentemente, afetam também as estratégias de podas de documentos e blocos durante o processamento das consultas, tanto do BMW, quanto do BMW-CS. Nota-se que o algoritmo apresenta um comportamento diferente para cada alteração no tamanho dos blocos. No entanto, como foi visto nesta seção, podemos ter uma redução do número total de blocos formados sem impactar negativamente o tempo de processamento. Unindo-se o uso de blocos de tamanho variável à estratégia de inserção de limiar inicial, conseguimos obter ganhos no tempo de processamento e redução no tamanho das *skiplists*.

5.1.2 Algoritmo BMW-CS

A Figura 5.6 mostra o estudo feito ao se utilizar blocos fixos com menos e mais documentos (64, 128 e 256), nas duas camadas da lista invertida do algoritmo BMW-CS. Todas as opções de implementação de blocos fixos exibidas e explicadas nesta seção utilizaram a estratégia de limiar de poda inicial. Unimos três variações em cada gráfico, escolhendo aquelas que definem melhor o comportamento do algoritmo quando utilizamos blocos de tamanho fixos. É importante lembrar que, assim como o BMW, quanto menos documentos há em cada bloco, mais entradas teremos nas *skiplists*. Isto aponta que haverá mais custos de memória, visto que mais blocos serão criados.

As Figuras 5.6(a), 5.6(c) e 5.6(e) indicam que blocos formados com menos documentos reduzem o tempo de processamento de consultas. No entanto, quando analisamos custos de memória somados a um tempo de processamento mediano, percebemos que menos entradas nas *skiplists* (Figura 5.7), entre as variações apresentadas, geram uma configuração melhor para o algoritmo. Observamos também que a diferença de tempo de processamento entre todas as variações expostas (Figura 5.6) diferem em poucos milisegundos. Isto mostra que podemos reduzir custos de memória sem causar muito impacto no tempo de processamento do algoritmo. Na pior hipótese da formação de blocos com poucos documentos, por exemplo, 64 documentos em ambas camadas, temos no total quase 24 milhões de entradas nas *skiplists*, enquanto que para blocos com 256, em ambas camadas, temos aproximadamente 6 milhões de entradas. A opção de implementar blocos fixos com 128 documentos nas duas camadas acarreta em custos medianos em relação ao número total de blocos (média aproximada de 11 milhões) formados para cada porcentagem da primeira camada.

Na aplicação da abordagem de blocos variáveis, mudamos o tamanho dos blocos de 64 até 1024 documentos nas duas camadas e analisamos o desempenho do algoritmo para cada porcentagem de variação da primeira camada, para top-10 e top-1000 respostas. Apresentamos na Figura 5.8 três opções de implementação com blocos variáveis, com a primeira, segunda ou ambas camadas variáveis. Todas as implementações foram comparadas ao algoritmo BMW-CS Original que apresenta blocos fixos com 128 documentos e adota a estratégia de limiar inicial. Os gráficos da figura mostram que não obtivemos nenhum ganho tanto para o cenário de top-10 como top-1000 respostas. A Tabela 5.2 mostra a média de tempo entre as variações de 18% e 50% da primeira camada para todas

(a) Top-10

(b) Top-1000

(c) Top-10

(d) Top-1000

(e) Top-10

(f) Top-1000

Figura 5.6: Desempenho do algoritmo BMW-CS variando o número de blocos fixos nas duas camadas do índice.

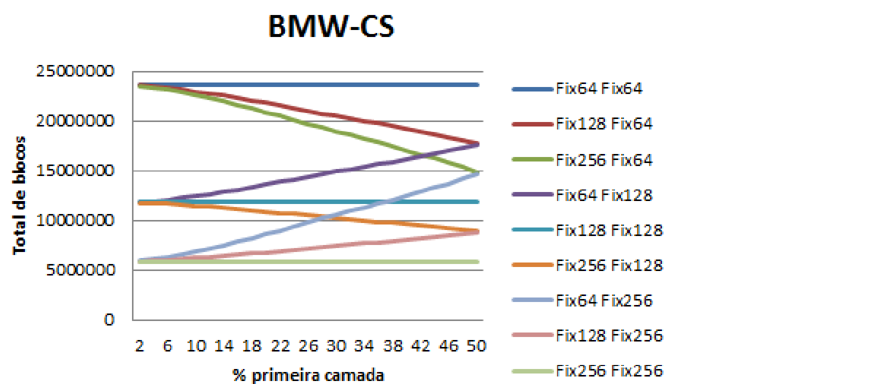


Figura 5.7: Número total de entradas das *skiplists* para cada porcentagem da primeira camada - blocos fixos de 64, 128 e 256 documentos.

as opções de implementação no cenário de top-1000 respostas. Constata-se que os tempos de processamento obtidos pelas abordagens são bem similares, e observa-se também que com o uso apenas da estratégia de limiar inicial podemos obter uma redução maior de tempo sem que seja necessário utilizar a abordagem de blocos variáveis.

	1ª camada variável	**2ª camada variável**	**1ª e 2ª camadas variáveis**
Mín 64 Máx 256	48,47	48,88	49,18
Mín 64 Máx 512	48,76	49,65	50,41
Mín 64 Máx 1024	49,00	50,12	51,35
BMW-CS Original: 57,94			
BMW-CS Original c/ limiar: 47,82			

Tabela 5.2: Blocos Variáveis - Média do tempo de processamento em milisegundos calculada a partir da variação de 18% da primeira camada para top-1000 respostas.

Tendo em vista que o tempo de processamento entre as estratégias de blocos variáveis não apresenta grandes variações, podemos levar em consideração, como nas abordagens anteriores, reduzir custos de memória. A Figura 5.9 faz um comparativo do total de blocos entre todas as opções de implementação, inclusive a utilizada no algoritmo BMW-CS original. Ainda na figura, notamos que o algoritmo BMW-CS fixo com 128 documentos configura a pior hipótese de implementação, visto que o mesmo totaliza a média de aproximadamente 11.8 milhões de blocos formados. No entanto, a estratégia de variar as duas camadas com o mínimo 64 e máximo 1024 documentos reduz em aproximadamente 78% o total de blocos, quando o comparamos com o algoritmo original. A opção de implementação de variar as duas camadas com o mínimo de 64 e máximo 256 documentos gera uma redução mediana aproximada de 43% em relação ao BMW-CS original, sem mudanças significativas no tempo de processamento. Por exemplo, em nossos experimentos, o tempo de processamento (Tabela 5.2) do algoritmo BMW-CS original, que utiliza a estratégia de limiar inicial, é de $47,82$(ms), enquanto que temos um tempo de $49,18$(ms) ao variamos as duas camadas com o mínimo de 64 e máximo 256 documentos.

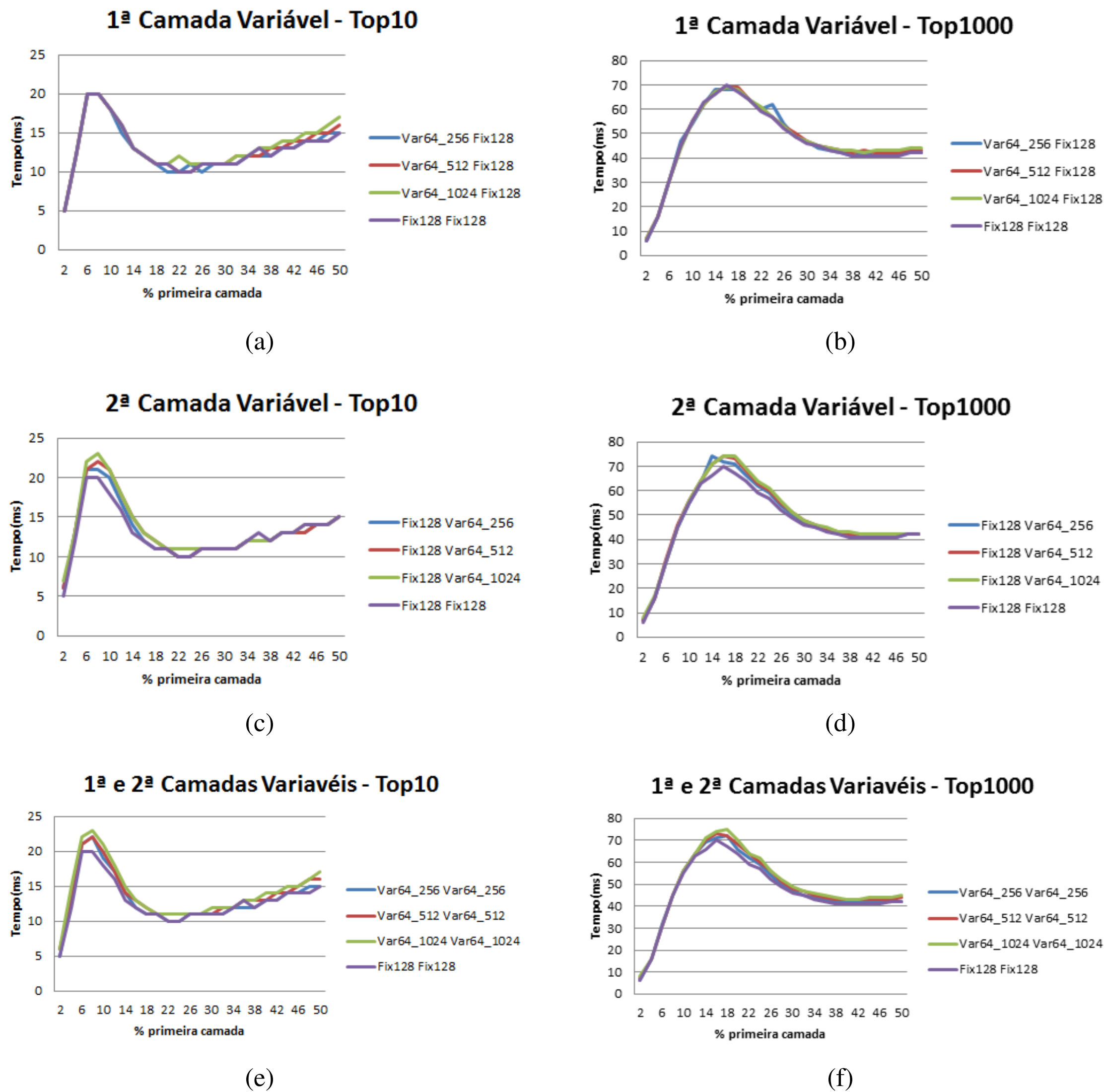


Figura 5.8: Desempenho do algoritmo BMW-CS variando o número de blocos nas duas camadas do índice.

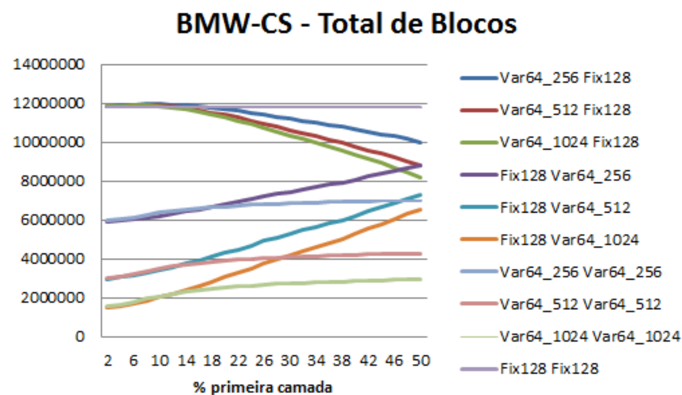


Figura 5.9: Número total de entradas das skiplists para cada porcentagem da primeira camada - blocos variáveis.

Algoritmo 2 Pseudocódigo da geração de *skiplists* de blocos variáveis.

```
1: Let L be the index
2: Let VetMilScore be the buffer of millesimal
3: for each 0 ≤ i ≤ |L| do
4:     sizeBlock ← 128; //Minimum value
5:     numDoc ← 0;
6:     load(lista[i]);
7:     if getDoc(lista[i], doc) then
8:         numDoc++;
9:     end if
10:    score ← BM25(lista[i], doc);
11:    maxScoreBlock ← score;
12:    maxDocs ← numDoc;
13:    previousFlag ← false;
14:    while getDoc(lista[i], doc) do
15:        score ← BM25(lista[i], doc);
16:        numDoc++;
17:        if (maxScoreBlock < score) then
18:            maxScoreBlock ← score;
19:        end if
20:        //Creating of fixed blocks
21:        if (numDoc % sizeBlock = 0) then
22:            if (maxScoreBlock > VetMilScore[lista[i]]) then
23:                if (indice! =0 && previousFlag) then
24:                    updateSkip(bufferSkip, previousDoc, previousMaxScoreBlock);
25:                end if
26:                updateSkip(bufferSkip, doc, maxScoreBlock);
27:                maxScoreBlock ← 0;
28:                previousFlag ← true;
29:                maxDocs ← 0;
30:            else
31:                if (maxDocs == 1024) then //Maximum value
32:                    updateSkip(bufferSkip, doc, maxScoreBlock);
33:                    maxScoreBlock ← 0;
34:                    previousFlag ← true;
35:                    maxDocs ← 0;
36:                else
37:                    previousFlag ← false;
38:                end if
39:            end if
40:            previousDoc ← doc
41:            previousMaxScoreBlock ← maxScoreBlock
42:        end if
43:    end while
44:    if (numDoc ! = 0) then
45:        updateSkip(bufferSkip, doc, maxScoreBlock);
46:    end if
47:    createSkip(bufferSkip, lista[i]);
48: end for
```

Capítulo 6

Conclusão

Este trabalho abordou estratégias de poda e mudanças na estrutura de índices dos algoritmos de processamento de consultas de busca textual: BMW e BMW-CS.

Os resultados obtidos mostram que estimar limiares iniciais para descartes de documentos pode melhorar o desempenho do algoritmo BMW e suas variantes tanto para o cenário de top-10 como top-1000 respostas. Quando estimamos o limiar ótimo como referência, obtemos uma redução significativa no tempo de processamento para o algoritmo BMW para blocos fixos de 64, 128 e 256 documentos. Os maiores ganhos no uso da estratégia desenvolvida de décimo/milésimo maior *score* estão relacionados a blocos fixos de tamanho 64, onde temos reduções aproximadas de 5,2% e 7,8% para top-10 e top-1000 respostas, respectivamente. Com a mesma estratégia, o algoritmo BMW-CS obteve maiores ganhos, com reduções de aproximadamente 17%. Observa-se, para este algoritmo, um tempo de processamento similar entre as implementações de blocos fixos com 64, 128 e 256 documentos. Os resultados também mostram que a implementação de blocos fixos com 256 entradas reduz em aproximadamente 50% o total de blocos formados. Nos dois algoritmos, nota-se que os maiores impactos em relação ao tempo de processamento são percebidos na medida em que o número de respostas retornadas aumenta. Ou seja, para top-k respostas, quanto maior for o valor de k, maior será o impacto na redução do tempo de processamento do algoritmo com a estratégia de limiar de poda inicial, comparado ao algoritmo que utiliza limiar inicial igual a zero.

A implementação da estratégia de variação do tamanho dos blocos da estrutura de índices não acarretou em melhoras no tempo de processamento, porém proporcionou reduções significativas em relação ao número de entradas das *skiplists*. Apesar da idéia de

variar blocos ser uma proposta que obtém ganhos relacionados a redução do número de entradas de todas as estruturas *skiplists*, ela apresenta uma dinâmica que pode aumentar consideravelmente o tempo de processamento de consultas. Tendo em vista este problema, executamos a proposta em conjunto com a estratégia de limiar de poda inicial, para o algoritmo BMW e BMW-CS.

No algoritmo BMW, foram implementadas abordagens que variavam blocos com mais documentos (mínimo 16 e máximo 1024), e também blocos onde o tamanho máximo era de 128 ou 256 documentos. Ao utilizar blocos com mais documentos, observou-se um aumento no tempo de processamento em relação ao *baseline*, contudo reduziu-se em até aproximadamente 80% o número de entradas do total de *skiplists*. As melhores propostas foram as que utilizam blocos variáveis com até 128 e 256 documentos, pois as mesmas reduzem ou mantêm o tempo de processamento do *baseline*. Visto isso, analisamos que a melhor configuração emprega blocos com o máximo de 256 entradas, pois mantivemos o mesmo tempo de processamento do algoritmo original e reduzimos em até 35% o número total de blocos criados.

Para o algoritmo BMW-CS, ponderamos algumas implementações utilizando blocos fixos com 64, 128 e 256 documentos. Verificamos que a adoção de blocos fixos com 128 documentos, nas duas camadas, forma uma média de quase 12 milhões de blocos enquanto que a execução do algoritmo com blocos fixos de 256 documentos, também nas duas camadas, produz um tempo de processamento similar ao original, porém apresenta aproximadamente 6 milhões de blocos. Isto mostra que utilizar blocos fixos de 256 reduz em uma média de aproximadamente 50% o total de entradas das *skiplists*.

As implementações de blocos variáveis realizadas no algoritmo BMW-CS acarretaram em poucos ganhos de tempo de processamento. Os resultados expostos em relação ao tempo de processamento foram gerados a partir das implementações de blocos variáveis que adotam a estratégia de décimo/milésimo como limiar inicial. Tais resultados mostram que implementar blocos fixos, com 128 documentos em cada camada, empregando a estratégia de limiar inicial, permanece sendo a melhor abordagem. Isto acontece pois a mesma tem uma redução de tempo superior à proposta na variação de blocos. Tendo em vista apenas o total de blocos formados, temos que a opção de implementação de variar as duas camadas com o mínimo de 64 e máximo 256 documentos gera uma redução mediana aproximada de 43% em relação ao BMW-CS original, sem mudanças significativas

no tempo de processamento.

Todos os experimentos realizados visaram adotar cenários próximos aos que são utilizados nos algoritmos originais tentando reduzir ou mesmo manter o tempo de processamento de consultas destes algoritmos. Os resultados obtidos mostram que as estratégias propostas em conjunto geram ganhos significativos ao algoritmo BMW e seu variante BMW-CS, principalmente na redução do total de dados armazenados nas *skiplists*.

6.1 Trabalhos Futuros

Como trabalho futuro, seria interessante estudar o impacto das técnicas aqui implementadas em cenários onde o índice do sistema de busca não cabe em memória. Em tais situações, ganhos de espaço nas *skiplists* podem tornar-se mais relevantes e ter impacto positivo no tempo de processamento, uma vez que *skiplists* menores têm maior chance de serem colocadas completamente em memória.

Referências Bibliográficas

[1] ANH, V. N., AND MOFFAT, A. Index compression using fixed binary codewords. In *Proceedings of the 15th Australasian database conference-Volume 27* (2004), Australian Computer Society, Inc., pp. 61–67.

[2] ANH, V. N., AND MOFFAT, A. Pruned query evaluation using pre-computed impacts. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval* (2006), ACM, pp. 372–379.

[3] BAEZA-YATES, R., RIBEIRO-NETO, B., ET AL. *Modern information retrieval*. ACM press New York, 2011.

[4] BLANCO, R., AND BARREIRO, Á. *Static pruning of terms in inverted files*. Springer, 2007.

[5] BRODER, A. Z., CARMEL, D., HERSCOVICI, M., SOFFER, A., AND ZIEN, J. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management* (2003), ACM, pp. 426–434.

[6] BUCKLEY, C., AND LEWIT, A. F. Optimization of inverted vector searches. In *Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval* (1985), ACM, pp. 97–110.

[7] CARMEL, D., COHEN, D., FAGIN, R., FARCHI, E., HERSCOVICI, M., MAAREK, Y. S., AND SOFFER, A. Static index pruning for information retrieval systems. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval* (2001), ACM, pp. 43–50.

[8] CERI, S., BOZZON, A., BRAMBILLA, M., VALLE, E. D., FRATERNALI, P., AND QUARTERONI, S. *Web Information Retrieval.* Springer Publishing Company, Incorporated, 2013.

[9] DE MOURA, E. S., DOS SANTOS, C. F., FERNANDES, D. R., SILVA, A. S., CALADO, P., AND NASCIMENTO, M. A. Improving web search efficiency via a locality based static pruning method. In *Proceedings of the 14th international conference on World Wide Web* (2005), ACM, pp. 235–244.

[10] DING, S., AND SUEL, T. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval* (2011), ACM, pp. 993–1002.

[11] ELIAS, P. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on 21*, 2 (1975), 194–203.

[12] GULLI, A., AND SIGNORINI, A. The indexable web is more than 11.5 billion pages. In *Special interest tracks and posters of the 14th international conference on World Wide Web* (2005), ACM, pp. 902–903.

[13] HEMAN, S. Super-scalar database compression between ram and cpu-cache. *Master's Thesis. University of Amsterdam. Amsterdam, The Netherlands* (2005).

[14] NGUYEN, L. T. Static index pruning for information retrieval systems: A posting-based approach. In *Proceedings of LSDS-IR, CEUR Workshop* (2009), pp. 25–32.

[15] ROBERTSON, S. E., AND WALKER, S. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval* (1994), Springer-Verlag New York, Inc., pp. 232–241.

[16] ROSSI, C., DE MOURA, E. S., CARVALHO, A. L., AND DA SILVA, A. S. Fast document-at-a-time query processing using two-tier indexes. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval* (2013), ACM, pp. 183–192.

[17] SALTON, G., WONG, A., AND YANG, C.-S. A vector space model for automatic indexing. *Communications of the ACM 18*, 11 (1975), 613–620.

[18] SCHOLER, F., WILLIAMS, H. E., YIANNIS, J., AND ZOBEL, J. Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval* (2002), ACM, pp. 222–229.

[19] STROHMAN, T., AND CROFT, W. B. Efficient document retrieval in main memory. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval* (2007), ACM, pp. 175–182.

[20] TROTMAN, A. Compressing inverted files. *Information Retrieval 6*, 1 (2003), 5–19.

[21] TURTLE, H., AND FLOOD, J. Query evaluation: strategies and optimizations. *Information Processing & Management 31*, 6 (1995), 831–850.

[22] VO, A. N., AND MOFFAT, A. Compressed inverted files with reduced decoding overheads. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval* (1998), ACM, pp. 290–297.

[23] WILLIAMS, H. E., AND ZOBEL, J. Compressing integers for fast file access. *The Computer Journal 42*, 3 (1999), 193–201.

[24] YAN, H., DING, S., AND SUEL, T. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on World wide web* (2009), ACM, pp. 401–410.

[25] ZHANG, J., LONG, X., AND SUEL, T. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th international conference on World Wide Web* (2008), ACM, pp. 387–396.

[26] ZHENG, L., AND COX, I. J. Document-oriented pruning of the inverted index in information retrieval systems. In *Advanced Information Networking and Applications Workshops, 2009. WAINA'09. International Conference on* (2009), IEEE, pp. 697–702.

[27] ZOBEL, J., AND MOFFAT, A. Inverted files for text search engines. *ACM computing surveys (CSUR) 38*, 2 (2006), 6.

[28] ZUKOWSKI, M., HEMAN, S., NES, N., AND BONCZ, P. Super-scalar ram-cpu cache compression. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on* (2006), IEEE, pp. 59–59.