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

PROCESSAMENTO EFICIENTE DE CONSULTAS EM SISTEMAS DE BUSCA.

Caio Moura Daoud

Novembro de 2016

Manaus - AM

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

PROCESSAMENTO EFICIENTE DE CONSULTAS EM SISTEMAS DE BUSCA.

Caio Moura Daoud

Tese de Doutorado 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 Doutor em Informática.

Orientador: Prof. Edleno Silva de Moura, D.Sc.

Novembro de 2016

Manaus - AM

Ficha Catalográfica

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

Daoud, Caio Moura
D211p Processamento eficiente de consultas em sistemas de busca / Caio Moura Daoud. 2016
79 f.: il. color; 31 cm.

Orientador: Edleno Silva de Moura
Tese (Doutorado em Informática) - Universidade Federal do Amazonas.

1. Processamento de consultas. 2. Máquina de Busca. 3. Recuperação de informação. 4. MétodoWaves. I. Moura, Edleno Silva de II. Universidade Federal do Amazonas III. Título

PROCESSAMENTO EFICIENTE DE CONSULTAS EM SISTEMAS DE BUSCA.

Caio Moura Daoud

TESE 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 DOUTOR EM INFORMÁTICA.

Aprovada por:

_____

Prof. Edleno Silva de Moura, D.Sc.

NOVEMBRO DE 2016

MANAUS, AM – BRASIL

Resumo

Trabalhos na literatura propõem diferentes técnicas para processamento de consultas em sistemas de busca. Esses sistemas são capazes de buscar informação relevante dentro de grandes coleções de dados e estão entre as principais formas de se obter informações na Internet. A popularização desses sistemas, associada ao crescimento constante de dispositivos eletrônicos para armazenamento e produção de informação, impulsionam pesquisas não apenas em relação à qualidade da resposta final fornecida aos usuários mas também com relação à redução no tempo de processamento de consultas. O foco principal deste trabalho é o desenvolvimento de soluções que reduzam o tempo de processamento de consultas sem afetar a qualidade de respostas fornecidas por sistemas de busca. Como usuários tipicamente estão interessado apenas em um determinado número de respostas do topo do ranking, estudamos o cenário mais comum onde busca-se computar rapidamente apenas os k documentos de maior escore dentre os que atendem às consultas dos usuários.

São propostos, implementados e avaliados dois novos métodos de processamento de consultas, o método *Block Max WAND with Candidate Selection and Preserving Top-K Results* (BMW-CSP) e o método Waves. Os dois métodos utilizam uma abordagem documento-a-documento e índices em multi-camadas como base para reduzir o tempo de processamento de consultas.

O método BMW-CSP é uma extensão do método BMW-CS, um método proposto anteriormente na literatura. Apesar de muito eficiente, o BMW-CS apresenta a desvantagem de não garantir a corretude dos resultados do topo das respostas em sistemas de busca por poder descartar documentos que estariam originalmente entre as melhores respostas. O método BMW-CSP modifica o BMW-CS para resolver o problema, tornando-se um método que calcula corretamente o escore de todos os documentos. Tanto o método BMW-CS quanto o BMW-CSP apresentam como desvantagem a necessidade de utilizar

memória extra para armazenar resultados parciais obtidos pelos métodos durante o processamento de consultas. Estudando mais a fundo o problema, propôs-se aqui um novo algoritmo que não requer tal expaço extra de armazenamento, o algoritmo Waves.

O método Waves realiza passadas sucessivas pelas diversas camadas dos índices. Cada passagem foi denominada aqui de *wave* (onda em inglês), o que deu origem ao nome do método. Cada passagem sobre o índice é numerada e dada uma i-ésima passagem, ela processa o índice apenas da i-ésima camada em diante. Após cada passagem, o algoritmo faz uma verificação para saber se já se pode garantir que os k maiores escores de documentos já foram computados corretamente. Se houver garantia, o algoritmo para o processamento. Do contrário, o algoritmo executa uma nova passagem no índice até que o resultado correto seja matematicamente garantido. Os experimentos realizados com diferentes bases e cenários indicam que os dois novos métodos podem processar consultas até duas vezes mais rápido que os principais métodos propostos anteriormente na literatura.

PALAVRAS-CHAVE: Processamento de consultas, Máquina de Busca, Recuperação de informação.

Abstract

Search systems have been one of the main forms of locating and retrieving information in digital environments in recent decades. They are present in a large number of applications, such as web search engines and e-commerce systems. Users of these systems more often than not have very specific information needs, only being satisfied with a few, highly relevant results. Due to this behavior, part of the recent research effort related to search systems aims to reduce computational costs to compute the top results of queries, which are the ones usually presented to most users.

In this thesis, we study the problem of computing the top k results of a ranking in search engines. We present two novel document-at-a-time algorithms for fast computing of top-k query results in search systems, named as *Block Max WAND with Candidate Selection and Preserving Top-K Results* (BMW-CSP) and Waves. Both algorithms use multi-tier indexes for reducing the computational time required for processing queries. BMW-CSP is an extension of BMW-CS, a method previously proposed in the literature. Although very efficient, BMW-CS does not guarantee the preservation of the top-k results for a given query. Algorithms that do not preserve the top results may reduce the quality of ranking results in search systems. BMW-CSP extends BMW-CS to ensure that the top-k results will have their rankings preserved. In the experiments we performed for computing the top-10 results, the final average time required for processing queries with BMW-CSP was lesser than the ones required by the baselines adopted. As with BMW-CS, the price paid by BMW-CSP, when compared to other document-at-a-time methods, is extra memory required to store partial scores of documents.

Further studying the problem of query processing, we then proposed Waves. It performs successive tentative evaluations of results which we call waves. Each wave traverses the index, starting from a specific tier level i. Each wave i may insert only those documents that occur in that tier level into the answer. After processing a wave, the al-

gorithm checks whether the answer achieved might be changed by successive waves or not. A new wave is started only if it has a chance of changing the top-k scores. We show through experiments that such lazy query processing strategy results in smaller query processing times when compared to previous approaches proposed in the literature. When compared to BMW-CSP, Waves presents the advantage of not requiring extra memory to store partial scores. We present experiments to compare the performance of Waves to BMW-CSP and to other state-of-the-art document-at-a-time query processing methods that preserve top-k results. These experiments indicate that the method can be an effective alternative algorithm for computing top-k results.

KEY-WORDS:: Query processing, BMW

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Máquinas de busca são sistemas que procuram informações relevantes em uma coleção de documentos de acordo com as necessidades do usuário. Nesses sistemas o usuário especifica suas necessidades de informação por meio de uma consulta, comumente descrita por um conjunto pequeno de palavras-chave. Com esse conjunto de palavras-chave, também chamadas de termos, o sistema busca por documentos que possam ser relevantes para o usuário. O resultado desse processo é uma lista de documentos ordenados de acordo com uma estimativa de relevância de cada documento como resposta à consulta feita pelo usuário.

O resultado para uma dada consulta utiliza em geral um modelo de recuperação de informação (RI), como o Modelo de Espaço Vetorial [27] ou o BM25 [24], para calcular o escore (similaridade entre o documento e a consulta, que pode ser vista como uma estimativa da relevância do documento como resposta à consulta) de cada documento e então ordená-los. Os documentos com maior escore são selecionados para compor o resultado que será retornado para o usuário ou utilizado por outro processo de ordenação mais complexo [9]. Nos dois casos, o processamento dos documentos para que se obtenha essa ordenação é parte determinante no custo final do processamento de consultas.

O constante crescimento no número de meios eletrônicos para armazenamento de informação junto com a popularização de sistemas de busca, trazem consigo a necessidade por soluções que possam reduzir os custos para processamento de consultas. Apesar dos índices utilizados por sistemas de busca proverem um acesso rápido aos documentos indexados, o custo de se processarem consultas cresce conforme o tamanho da coleção. Visando diminuir tal custo, projetistas têm empregado diversas estratégias e técnicas para

aumentar a escalabilidade e reduzir o tempo gasto para processar cada consulta submetida aos sistemas de busca.

Esta tese teve como objetivo o estudo de algoritmos para processamento de consultas em sistemas de busca, visando investigar formas de aprimorar os melhores algoritmos encontrados. O objetivo principal do estudo é propor, implementar e avaliar algoritmos que reduzam o custo computacional necessário para processar consultas em máquinas de busca sem afetar a qualidade da resposta.

1.1 Contribuições

Nesta tese são apresentados dois novos métodos de processamento de consultas. O primeiro método, chamado BMW-CSP [11], modifica o BMW-CS proposto por Rossi et al. [25] para garantir o processamento de consultas sem a perda de resultados ocorrida no BMW-CS. A melhoria foi obtida sem perda significativa no desempenho. Tanto o método BMW-CS quando o BMW-CSP apresentam como desvantagem a necessidade de espaço extra para armazenar resultados parciais durante o processamento de consultas. Esse ponto motivou o desenvolvimento de um novo método de processamento também inspirado nos dois primeiros, que foi chamado de Waves. Em nossos experimentos, o método Waves obteve desempenho melhor que o BMW-CSP com a vantagem de não exigir uso de espaço adicional para armazenar resultados parciais na memória. Quando comparado aos métodos propostos anteriormente na literatura, o método Waves chega a ser duas vezes mais rápido que o melhor *baseline* encontrado. Também é apresentado neste trabalho um estudo e uma avaliação dos melhores métodos de processamento de consulta encontrados na literatura.

1.2 Organização do trabalho

Este trabalho está estruturado da seguinte forma. No Capítulo 2, são apresentados os conceitos básicos de um sistema de busca. No Capítulo 3, são apresentados alguns métodos de processamento de consulta encontrados na literatura. Detalhes dos novos métodos, BMW-CSP e Waves, são apresentados no Capítulo 4. No Capítulo 5 são apresentados os experimentos desenvolvidos para avaliar os métodos apresentados, demonstrando os

ganhos alcançados com os novos métodos quando comparados com o melhor *baseline*. Ainda no mesmo capítulo são descritos os experimentos. O Capítulo 6 encerra este trabalho com a conclusão e direcionamento para trabalhos futuros.

Capítulo 2

Conceitos Básicos

Um documento é representado em um sistema de busca pelo conjunto de termos que o compõe. O conjunto de termos distintos dentro de uma coleção de documentos é chamado de vocabulário. Quando um usuário especifica uma consulta por meio de um conjunto de termos, o sistema de busca tem por objetivo encontrar os *top-k* documentos mais similares a essa consulta, com base nos termos presentes na consulta e nos termos presentes no documento.

A estrutura de índices invertidos é comumente utilizada para acelerar o processamento das consultas (Baeza-Yates & Ribeiro-Neto [6]). Nessa estrutura, para cada termo do vocabulário, é gerada uma lista invertida contendo os documentos onde o termo ocorreu, juntamente com alguma informação como a frequência do termo no documento ou a lista de posições onde o termo ocorreu nos documentos.

Neste trabalho, o identificador numérico de um termo será referenciado como termId, e o identificador numérico de um documento será referenciado como docId. Cada entrada em uma lista invertida corresponde a um par (docId, freq), onde freq é a frequência do termo no documento docId.

Todo o processo de geração do vocabulário e dos índices invertidos da coleção de documentos é um processo feito de modo off-line, chamado indexação. Após o término do processo de indexação, as informações geradas são disponibilizadas para que o sistema de busca utilize-as no processamento de consultas. O processamento de consultas consiste em encontrar, dentro da coleção, os documentos que tenham alguma similaridade com a consulta. Por meio do índice invertido, as listas invertidas dos termos da consulta são recuperadas. A partir dessas listas é gerado o conjunto de documentos similares. Ao

final do processo, os documentos do conjunto são retornados, ordenados por um valor de similaridade com base em algum modelo de RI (recuperação de informação). O resultado desse processo é conhecido como *ranking*.

Segundo Baeza et al. [5], 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. Um dos modelos probabilístico mais tradicionais na área de RI é o Modelo de Espaço Vetorial proposto por Salton et al. [27], onde os documentos são representados como vetores, e o tamanho do vocabulário é a dimensão dos vetores. Dois documentos têm seu valor de similaridade determinado pela fórmula do cosseno, onde documentos idênticos têm valor de similaridade 1 e documentos completamente diferentes similaridade 0. Outro modelo de similaridade popular na literatura é o BM25 [24]. 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 6 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, $|D|$ 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 $K1 = 2$, conforme Baeza et al. [5].

2.1 Organização do índice invertido

A escolha de qual algoritmo será utilizado no processamento de consultas depende da forma como o índice invertido está organizado. Em grandes coleções, é comum empregar métodos de compressão sobre as entradas do arquivo invertido [16, 31, 34]. Isso diminui o espaço necessário para armazenar o índice, porém dificulta o acesso aleatório sobre as entradas da lista. Existem duas formas principais de organização de um índice invertido: índices ordenados por frequência e índices ordenados por documento (Grossman et. al [15]).

Em índices ordenados por frequência, cada lista invertida tem seus documentos armazenados em ordem decrescente de frequência, e documentos com mesmo valor de frequência são ordenados em ordem crescente de acordo com seu identificador numérico docId. Para economizar espaço de armazenamento é comum registrar apenas o valor de frequência e a quantidade de documentos com a mesma frequência, e em seguida a lista de documentos ordenados por docId. Alguns trabalhos propuseram que ao invés de se armazenar a frequência, seria possível armazenar diretamente o valor de similaridade do termo com o documento pré-computado dentro da lista invertida (Anh e Moffat [2]). Isso evita que os cálculos de similaridade sejam feitos durante o processamento de consultas.

Em índices ordenados por documento, cada lista invertida tem seus documentos armazenados em ordem crescente de documento. Para cada entrada na lista, é armazenado um par (docId, freq). Ao invés de armazenar o valor do docId, costuma-se armazenar apenas a diferença do docId do documento atual na lista para o seu anterior. Isso reduz os valores inteiros a serem comprimidos, melhorando a taxa de compressão e consequentemente reduzindo o tamanho físico do índice.

Em listas invertidas ordenadas por documento é utilizada uma estrutura, conhecida como *skiplist*, para acelerar o acesso aleatório sobre as entradas das listas invertidas. Para cada lista invertida é gerado uma *skiplist* que armazena ponteiros para determinadas entradas dentro da lista. Isso permite que um algoritmo de processamento utilize as *skiplists* para saltar entradas desnecessárias dentro de uma lista invertida sem a necessidade de acessar ou descomprimir essas entradas.

Neste trabalho, os métodos propostos e os *baselines* utilizam a estrutura de índices ordenados por documento e cada lista invertida possui uma *skiplist* associada, contendo uma entrada para cada bloco de 128 documentos.

2.2 Processamento de consultas

De acordo com a maneira como o índice invertido está organizado, existem duas abordagens principais para processar consultas: (i) processamento termo-a-termo (TAT), (Strohman e Croft [32]), que utiliza índices ordenados por frequência e (ii) processamento documento-a-documento (DAD), (Ding e Suel [14]), que utiliza índices ordenados por documento.

Algoritmos de processamento termo-a-termo percorrem as listas invertidas sequencialmente. Como um documento pode ocorrer em qualquer posição das listas invertidas, durante o processamento é criado um acumulador para cada novo documento avaliado, a fim de armazenar o valor de similaridade entre o documento e a consulta. O valor de similaridade é calculado parcialmente ao longo do processo e apenas no final é possível obter o valor completo de similaridade entre um documento e a consulta. Isso faz com que algoritmos de processamento termo-a-termo utilizem uma quantidade consideravelmente alta de acumuladores durante o processamento de uma consulta. Para diminuir o tempo de processamento, é possível aplicar métodos de poda [22] onde o processamento é interrompido quando se tem certeza de que o conjunto dos *top-k* documentos de resposta já foi obtido. Como as listas invertidas estão ordenadas por frequência, é possível determinar a contribuição máxima que um documento teria em determinada lista invertida. Isso torna possível calcular um limiar de poda que seja capaz de terminar o processo sem alterar o conjunto de resposta.

No processamento documento-a-documento, as listas invertidas são percorridas em paralelo, permitindo que um documento selecionado tenha seu valor completo de similaridade avaliado. Isto evita que seja necessário manter uma lista de acumuladores muito grande, bastando armazenar o conjunto de documentos de maior valor de similaridade encontrados no momento, reduzindo o consumo de memória do método. Utilizando a estrutura de *skiplists* é possível processar de maneira eficiente consultas conjuntivas, onde um documento de resposta deve conter todos os termos da consulta. Porém, como as listas invertidas não estão ordenadas por frequência, não é possível aplicar mecanismos de poda sobre consultas disjuntivas, onde não é necessário que um documento contenha todos os termos da consulta. Para contornar esse problema, diversos trabalhos foram propostos com o objetivo de diminuir a quantidade de entradas avaliadas durante um processamento documento-a-documento, por meio da estrutura de *skiplists* e técnicas de pivotação.

Os algoritmos de processamento podem ainda ser classificados em exatos e aproximados. Algoritmos de processamento exato garantem que o conjunto de respostas contenha todos os documentos de maior valor de similaridade com a consulta, dispostos na ordem correta segundo esse valor. Algoritmos aproximados não garantem que o conjunto de respostas retornado seja o mesmo retornado por um método exato, sendo que nesses casos é possível que um documento que tenha valor de similaridade suficiente para colocá-lo entre os *top-k* documentos retornados, fique fora da resposta por não ter sido avaliado. A maioria dos algoritmos aproximados buscam melhorar a eficiência no processamento de consultas por meio de uma poda mais agressiva, resultando na mudança do conjunto de resposta.

Atualmente o algoritmo de processamento de consultas termo-a-termo mais eficiente foi desenvolvido por Strohman e Croft [32]. Esse método foi superado pelo método de processamento documento-a-documento BMW proposto por Ding e Suel [14], o qual foi utilizado como baseline para os algoritmos apresentados neste trabalho e é tratado em detalhes na Seção 3.2. Resultados de experimentos com o método BMW e com os novos métodos são apresentados no Capítulo 5.

Capítulo 3

Trabalhos Relacionados

Nesta seção é apresentado um levantamento bibliográfico das técnicas de processamento de consultas. Os trabalhos apresentados estão organizados em três grupos de técnicas classificadas de acordo com a estratégia utilizada: (i) estratégias de cache: trabalhos têm como objetivo básico manter em cache as informações das consultas mais frequentes para que essas possuam custos mínimos quando voltarem a ocorrer; (ii) estratégias de poda: trabalhos que propõem estratégias para evitar o processamento de documentos que não possuem chance de compor o ranking de resposta; e (iii) estratégias multicamadas: nesses trabalhos, as listas invertidas são divididas em pelo menos duas camadas e o objetivo básico é evitar o processamento completo das listas invertidas de cada termo da consulta.

3.1 Cache

Segundo Baeza-Yates et al. [4], as técnicas de cache assumem que existe uma taxa de repetição na sequência de consultas que são submetidas ao sistema durante um período de tempo, possibilitando que, mesmo utilizando-se uma quantidade limitada de memória, possa ser feito um cache com taxas de acerto efetivas.

As técnicas de cache mais aplicadas são de resultados e termos. Cache de resultados mantém a página de resposta para uma dada consulta. Quando ocorre um acerto, o sistema retorna imediatamente o conjunto de respostas sem nenhum custo de processamento. Em cache de termos, o sistema mantém partes das listas invertidas de um conjunto de termos frequentes em memória para acelerar o processamento das consultas submetidas.

Retornar a resposta para uma consulta existente no cache de resultados é mais eficiente

que utilizar as listas invertidas do cache de termos para processá-la. Porém, a taxa de acerto do cache de resultados é significativamente menor que no cache de termos. Isso se deve ao fato de que muitos termos aparecem em diversas consultas, mas nem sempre seguidos dos mesmos termos. Através da análise de um histórico de consultas do Yahoo!, apresentado em [4], é mostrado que a taxa de acerto de um cache de respostas é limitada pela fração de consultas únicas presentes em uma máquina de busca, podendo atingir taxas de acerto em torno de 50%. Por outro lado, o cache de termos é limitado pela fração de termos distintos presentes nas consultas feitas, ficando com taxas de acerto em torno de 90%.

As técnicas de cache podem ser aplicadas em mais de um nível do processamento de consulta. Saraiva & de Moura [28] propõem uma arquitetura utilizando cache de resultados combinado com cache de blocos de listas invertidas para melhorar o tempo de processamento e aumentar a capacidade de processar consultas em uma máquina de busca real. Os autores mostram que a utilização dessas duas estruturas ajuda a aumentar a quantidade de consultas simultâneas processadas em relação a uma arquitetura sem um esquema de cache.

O preenchimento do cache pode ser dividido entre estratégias estáticas e dinâmicas. Para o cache estático a estratégia é manter em cache um conjunto de itens que é processado periodicamente, podendo ser composto de respostas ou listas invertidas de termos comuns em consultas. A escolha das entradas que serão armazenadas no cache é baseada em informações históricas, como a lista de consultas submetidas durante um período de tempo. Já na estratégia de preenchimento dinâmico, os itens mudam de acordo com a sua utilização. Em geral aplica-se uma política de substituição de itens para definir quem deve ser retirado assim que o cache estiver cheio. A política mais adotada é a LRU, substituição do menos recentemente utilizado. Assim, o conteúdo de um cache dinâmico varia constantemente de acordo com a sequência de consultas feitas. Em [3, 4] foi feito um estudo comparativo entre caches estáticos e dinâmicos, e observou-se que as taxas de acerto em caches estáticos são significativamente maiores que qualquer tipo de cache dinâmico. Para explorar a correlação temporal das consultas e ainda manter informação sobre a distribuição das consultas presentes no histórico, foi estudada a combinação de cache estático com dinâmico, dividindo a memória disponível entre os dois. Como resultado, verificou-se que tal combinação resulta em uma melhora nas taxas de acerto.

Long e Suel [18] estudaram a utilização de um cache de interseção de listas invertidas, onde pares de termos que ocorrem com muita frequência em históricos de consulta são processados e armazenados em disco. Assim, quando uma consulta possui três termos, contendo um par de termos com a sua interseção previamente calculada, economiza-se o tempo que seria gasto para processar o terceiro termo. Este cache é combinado com os de termo e de resposta. A utilização de um cache de interseção de listas se justifica devido ao fato do cache de respostas ter um bom desempenho para consultas com até dois termos. Em consultas com mais de dois termos, a porcentagem de pares de termos que ocorrem juntos seguidos de outros termos é muito maior.

No trabalho apresentado em [1], Leonardi et al. tratam o problema de cache como um problema de cobertura de consultas, considerando que um documento pode ser relevante para uma grande quantidade de consultas. Através de uma análise sobre o histórico de consultas, informações estatísticas sobre a distribuição das consultas durante um período de tempo são coletadas para montar um cache de documentos relevantes. O problema é modelado utilizando-se um algoritmo de multi-cobertura estocástica. O objetivo é gerar um mapeamento de uma consulta para um conjunto de documentos considerados relevantes para a mesma. De modo que quando chega uma consulta, os seus k documentos mais relevantes possam ser encontrados através deste mapeamento. Quando não são encontrados, ocorre um miss no cache. O trabalho não faz comparação com cache de resultados e não leva em consideração ordenação de resultados, o que dificulta a aplicação em um sistema de busca onde a ordenação das respostas é crucial.

3.2 Poda

Persin et al. [22] apresentam um método de processamento de consultas sobre índices ordenados por frequência que diminui a quantidade de entradas lidas das listas invertidas dos termos de cada consulta. Nesse método, o acumulador de um documento só é alterado se a combinação da importância do termo na coleção com a frequência do termo nos documentos onde ocorre tiver peso suficiente para manter o documento no conjunto final de respostas. Quando os documentos de uma lista não possuem peso suficiente para modificar o conjunto de respostas final, a lista é descartada, reduzindo tempo de processamento.

Um método de poda é classificado como poda dinâmica quando as entradas são removidas de acordo com os termos da consulta. Nesses métodos, as entradas a serem removidas são calculadas sempre que a consulta é realizada. Quando a poda não depende da consulta e pode ser realizada de maneira off-line, é chamada de poda estática. Nesse caso, as entradas são removidas uma única vez das listas invertidas e cada consulta processada só utiliza as entradas remanescentes nas listas.

Blanco e Barreiro [7] desenvolveram um método de poda estática onde termos comuns, chamados *stop words*, são removidos do índice. Diferentemente de outros trabalhos, a lista invertida de um termo classificado como *stop word* é completamente removida do índice. Em outros trabalhos, como o apresentado por Carmel et al. [9], são apresentados métodos de poda estática onde partes das entradas da lista são removidas. Esse método utiliza a função de ordenação da máquina de busca para determinar a importância de cada lista invertida e identificar as entradas que podem ser removidas. Para isso, cada termo do vocabulário da coleção é submetido como uma consulta de apenas um termo. A partir do conjunto de resultados ordenado, é mantida uma fração de documentos do topo da resposta. Os documentos restantes são eliminados da lista invertida.

Moura et al. [13] apresenta uma variação do método proposto por Carmel et al. [9] capaz de reduzir em 60% do tamanho do índice. Além de permitir uma economia no custo de armazenamento do índice, o tempo médio de processamento de uma consulta cai 40%, com quase nenhuma perda de precisão. O método apresentado em [9] leva em consideração apenas informação individual dos termos do vocabulário, perdendo informação sobre termos que podem ocorrer em comum nas consultas. O método proposto em [13] tenta solucionar esse problema analisando a coocorrência entre termos dentro de documentos da coleção. Além do topo da lista invertida de cada termo t, preserva-se também, para cada termo $t_2 != t$, as entradas em t correspondentes a documentos que aparecem na lista de t_2, sempre que t_2 e t apareçam no mesmo contexto nos documentos correspondentes a essas entradas.

Nos métodos de poda baseada em termos, as entradas menos importantes da lista invertida são removidas, sem levar em consideração a importância relativa de uma entrada com outras do documento em listas diferentes. Poda baseada em documentos remove os termos menos importantes do documento, porém não considera a importância relativa entre uma entrada em relação a outras na mesma lista. Nguyen [20] propõe um método

que mistura poda baseada em termo e em documento com o objetivo de conciliar os defeitos de cada método, permitindo que uma entrada que teve um peso baixo em relação às demais entradas da lista invertida e obteve um peso alto em relação aos demais termos presente no documento possa ser preservada durante a poda, e vice-versa.

Zheng e Cox [35], desenvolveram um conjunto de estratégias para eliminar documentos da coleção baseando-se na premissa de que nem todos documentos são igualmente importantes. Os experimentos apresentados pelos autores mostram que em alguns cenários o método é competitivo, ou até melhor que métodos de poda baseados em entradas das listas, apesar da ideia de remover documentos da coleção ser contraintuitiva.

Em [17] utilizou-se aprendizado de máquina para classificar documentos em uma coleção como sendo de alta qualidade ou ordinários. Com esse método, é criado um índice contendo apenas documentos de alta qualidade. Os experimentos realizados mostram que esse método é capaz de manter bons valores de precisão removendo 50% das páginas da coleção de referência .GOV e 95% da páginas de uma coleção de páginas da máquina de busca SOGOU. O método utiliza apenas informações relacionadas aos documentos, sem considerar informações relativas a consultas. Apesar do bom desempenho, o autor não garante que o método funcionaria em um sistema de busca real com bilhões de páginas e consultas variadas.

Em [2], os autores apresentam um método de poda dinâmica utilizando um índice ordenado por impactos que são pré-computados durante a construção do índice. O método de poda dinâmica divide as listas em blocos de acordo com o impacto de cada entrada. Cada bloco é processado em um determinado modo. Os blocos com as entradas de maior impacto são processados em modo disjuntivo, onde qualquer documento é adicionado ao conjunto de acumuladores. Quando o algoritmo identifica que todos os documentos que possam estar presentes no *top-k* de resposta foram encontrados, o processamento passa a ser em modo conjuntivo, onde apenas documentos já presentes no conjunto de acumuladores têm seu valor de similaridade alterado. Quando necessário, as demais entradas são processadas em modo $refinado$, onde são alterados apenas os valores de similaridade dos documentos que serão retornados como resposta. As demais entradas da lista são descartadas, economizando-se assim tempo de processamento e transferência de disco nas leituras das listas invertidas. Os autores mostram que esse modelo de processamento de consultas atinge bons ganhos em throughput de consultas. É apresentada outra versão do

algoritmo em [2], chamada Método B, onde o *top-k* de resposta pode ser aproximado. No Método B, apenas uma porcentagem dos resultados obtidos na fase disjuntiva são utilizados na fase conjuntiva. Com essa modificação observa-se ganhos de velocidade, porém o *top-k* de resposta não tem garantias de que será exato.

Em [32] é proposta uma variação do método apresentado em [2] onde o índice é mantido em memória. Uma poda dinâmica é aplicada em cada fase do processamento com o objetivo de reduzir a quantidade de acumuladores necessários para obter o conjunto final de respostas sem ser necessário avaliar todos os candidatos. O processamento é feito sobre um índice invertido ordenado por impacto. Esse é o método de processamento TAT mais eficiente encontrado na literatura.

De acordo com [8], estratégias de processamento TAT são mais aplicáveis sobre coleções relativamente pequenas devido à grande quantidade de acumuladores necessários durante o processamento de uma consulta. DAD tem a vantagem de não utilizar muitos acumuladores, necessitando de pouca memória durante o processamento de uma consulta, além de explorar o paralelismo nas leituras de disco, uma vez que várias consultas são processadas ao mesmo tempo.

Os autores de [8] desenvolveram um dos trabalhos mais importantes, em processamento de consultas DAD, apresentados nos últimos anos. Propõem um método de processamento de consultas DAD conhecido como WAND (*Weak* AND ou *Weighted* AND) [8]. Nesse trabalho os autores propuseram a primeira estratégia de poda eficiente para processamentos de consultas em modo disjuntivo no índice DAD. O algoritmo utiliza um novo operador para reduzir o número de documentos avaliados durante o processamento de consultas.

No processamento DAD é criado um *heap* para manter os top-K documentos de maior escore em cada iteração do processador da consulta. O menor escore presente no *heap* é usado como um limiar de descarte, que pode ser usado para acelerar o processamento das consultas. Em cada iteração um documento tem seu escore computado, mas o documento só é inserido no *heap* se seu escore superar o atual limiar de descarte.

No método WAND, durante a indexação da coleção de documentos é armazenado o maior escore de cada lista invertida, chamado de *Max Score*. No exemplo da Figura 3.1 é apresentada uma lista invertida ordenada por documento e os escores correspondentes aos documentos. Nesse exemplo o *Max Score* armazenado durante a indexação tem valor

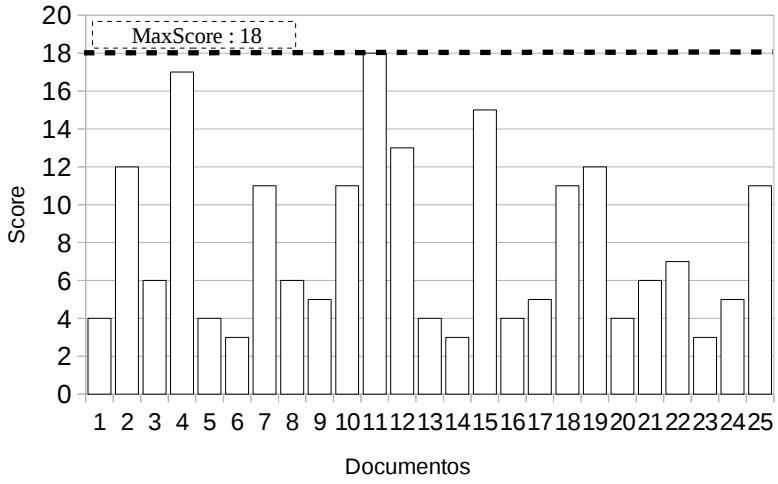

Figura 3.1: Max Score em uma lista invertida ordenada por documentos.

18 que corresponde ao documento 11. Durante o processamento, saber o escore máximo que um documento pode atingir é importante para evitar o custo de computar o escore de documentos que não tem chance de alterar o ranking de resposta. No processamento de uma consulta, documentos que estão sendo avaliados, chamados de pivôs, são analisados em duas etapas antes de serem inseridos no *heap*. Primeiro, obtém-se o *Max Score* deste documento, somando o *Max Score* de cada lista onde o pivô tem chance de ocorrer. Apenas quando o *Max Score* do pivô for maior que o limiar de descarte atual, é que a lista invertida de cada termo será acessada para que o documento pivô tenha seu escore real computado. Quando o *Max Score* não supera o limiar de poda, o documento pivô é descartado e um novo pivô é selecionado. Com essa abordagem o método WAND consegue descartar documentos durante o processamento de consultas, reduzindo do tempo final o custo de processar os documentos descartados.

Outro método muito importante de processamento de consultas DAD encontrado na literatura é o Block Max Wand (BMW). Proposto por Ding e Suel [14], tem como base a estratégia de poda proposta para o método WAND em [8]. A estratégia é descartar documentos durante o processamento com base em um cálculo de seu escore máximo, porém, os autores propõem uma solução otimizada que abrange principalmente uma modificação nas *skiplists*. Durante a indexação da coleção de documentos, para cada entrada da *skiplists* é armazenado o maior escore do bloco referenciado. A estratégia de poda é similar a adotada no método WAND com a vantagem de possuir um escore máximo mais próximo do escore real do documento.

Na Figura 3.2, é apresentado um exemplo do valor armazenado para cada bloco de uma lista invertida. A figura tem um exemplo de lista invertida ordenada por documento e considera uma entrada na *skiplist* para cada cinco documentos da lista. O peso máximo de um documento para toda a lista do exemplo tem valor 18 e para cada bloco valor $17, 11, 18, 12$ e 11 respectivamente.

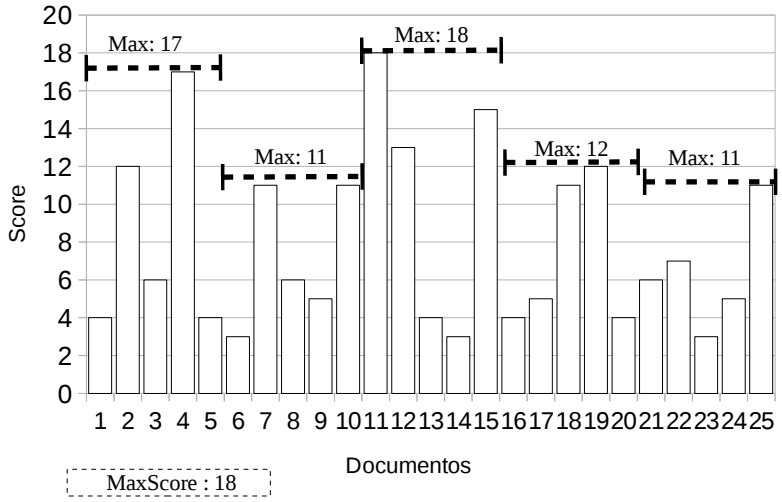

Figura 3.2: Block Max Score em uma lista invertida com blocos de 5 documentos

A poda é feita em dois níveis usando duas principais informações: (i) o maior escore de toda a lista invertida, conhecida como *Max Score*, que também é adotado no método WAND e (ii) o maior escore encontrado em cada bloco apontado pelas entradas da *skiplist*, conhecido como o *Block Max Score*.

Nesse método, se o *Max Score* for maior que o atual limiar de poda, é feito um movimento chamado de *shallow* que acessa apenas as entradas da *skiplist* para alinhar as listas invertidas sobre os blocos que, eventualmente, tem o documento pivô. Após o alinhamento, o algoritmo usa as informações armazenadas nas *skiplists* para calcular o *Block Max Score* do pivô, que é a soma do escore máximo de cada bloco onde o pivô tem chance de ocorrer. Somente quando esse valor for maior que o atual limiar de descarte a lista invertida de cada termo será acessada para que o documento pivô tenha seu escore real computado. Caso contrario, o documento é descartado, uma das listas será avançada e um novo documento pivô é selecionado. Assim, o método BMW descarta ainda mais documentos que não apresentam escore suficiente para serem inseridos no *ranking* de respostas, garantindo processamento de consulta mais rápido que o método Wand.

3.3 Multi-camadas

Quando um usuário faz uma consulta em um sistema de busca, geralmente está interessado apenas em um pequeno conjunto de documentos que sejam relevantes para sua consulta. Caso não os encontre nas primeiras páginas de resultado, os usuários tendem a mudar a consulta. Sendo assim, o processamento de todos os documentos de uma lista invertida para uma dada consulta pode acarretar em um grande desperdício computacional. Theobald et al. [33] desenvolveram um método de processamento de consultas para computar o conjunto aproximado dos k documentos de maior similaridade com a consulta sem a necessidade de processar todas as entradas das listas invertidas. A técnica consiste em processar o índice em camadas que começam com índices pequenos com entradas selecionadas segundo algum critério e terminam com camadas maiores.

O processamento em camadas pode ser feito de maneira sequencial (Baeza-Yates et al. [5]). Uma primeira camada é processada e caso o conjunto de documentos não seja suficiente, o processamento continua nas camadas seguintes até que o conjunto de respostas seja considerado satisfatório. Em um processamento em paralelo, todas as camadas são processadas ao mesmo tempo, caso uma camada forneça um conjunto de respostas satisfatório, o processamento nas demais camadas é suspenso. O ganho de desempenho se dá quando uma consulta consegue ser processada pela primeira camada, normalmente composta por um pequeno conjunto de documentos, e por isso é mais rápida para ser processada.

Risvik et al. [23] estudaram a divisão do índice em camadas, de modo que a maioria das consultas pudesse ser processada sem a necessidade de passar por todas as camadas do índice, diminuindo o tempo de processamento de uma consulta. Nesse trabalho, é criada uma arquitetura de índice, onde um documento é armazenado em determinada camada de acordo com o seu valor de impacto (peso atribuído à entrada em função do modelo de RI adotado). Cada documento fica em uma única camada. O índice é dividido em três camadas, sendo que a primeira possui um pequeno conjunto de documentos, a segunda uma quantidade maior e a terceira o resto dos documentos. Por causa da pequena quantidade de documentos na primeira camada, o processamento de consultas torna-se extremamente rápido. O autor utiliza um algoritmo para determinar se o processamento deve prosseguir para uma camada mais profunda, ou se o conjunto de respostas é satisfatório. O ganho em desempenho é dado por evitar que o processamento seja feito nas camadas mais profundas

onde está a maior quantidade de documentos. Apesar dos ganhos em desempenho, existe uma penalidade na qualidade das respostas, pois não existe a garantia de que o conjunto de respostas retornado seja o mesmo conjunto caso todo o índice fosse processado.

No trabalho apresentado em [21] é criado um índice com duas camadas: na primeira camada fica uma amostra do índice com os documentos mais importantes da coleção; na segunda camada, fica o índice completo. Essa amostra é um índice podado utilizando abordagens de poda de termos e documentos menos importantes. É proposto um algoritmo que determina se o conjunto de respostas retornado pelo primeiro índice é exatamente o mesmo que seria retornado pelo índice completo. Para isso é avaliada a fórmula de ordenação usada pelo sistema durante o processamento de consultas. Quando uma entrada não está no índice podado, é atribuída uma constante que garante o maior impacto que poderia ser atribuído ao documento caso fosse processado no índice completo. O método tem um bom desempenho, utilizando 10-20% do índice na primeira camada. [27] avalia o impacto da inserção de um cache no sistema, e mostra que a distribuição das consultas é alterada. As consultas com poucos termos, que seriam melhor tratadas pelo índice podado, são resolvidas pelo próprio cache, deixando assim as consultas mais difíceis para o índice podado processar e diminuindo o seu ganho. A combinação do cache com o índice podado aumenta o desempenho do sistema. O ganho em desempenho ocorre quando o processamento não é feito nos níveis mais profundos, os quais possuem uma quantidade maior de documentos para serem processados.

Em [14], onde é apresentado o método de poda BMW, os autores também apresentam o método MBMW. É a versão do BMW quando usado para processar índice em multi camadas. A diferença está no processo de indexação da coleção de documentos, onde para cada termo é gerado no mínimo duas listas invertidas. Essa divisão das listas favorece o método BMW por reduzir a distancia entre os pesos dos documentos de um mesmo bloco, permitindo que o *blockMaxScore* seja um valor mais próximo aos escores dos documentos do bloco.

A vantagem adquirida com a divisão das listas em camadas está na maneira como as entradas da lista são distribuídas entre as camadas. A divisão da lista se dá com base nos pesos das suas entradas, ficando as entradas de maior peso na primeira camada e as demais entradas nas camadas seguintes. Isso leva a uma melhor homogeneização dos pesos por camada, melhorando a qualidade representativa do *blockMaxScore* sobre as entradas do

bloco.

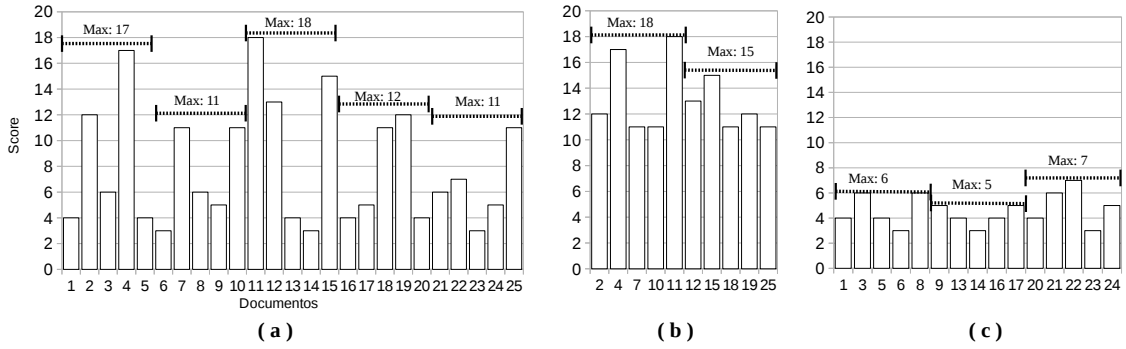

Figura 3.3: Divisão do índice inteiro, (a), em duas camadas, (b) de 40% e (c) de 60%. Em uma lista com blocos de 5 documentos

A Figura 3.3 apresenta um exemplo da divisão de uma lista invertida (a) em duas camadas (b) e (c). A lista possui 25 documentos e uma entrada na *skip list* para cada 5 documentos. Os valores de *blockMaxScore* obtidos são 11, 11, 12, 17 e 18 para a lista inteira (a). No exemplo em camadas apresentado na figura, a primeira camada (b) possui 40% das entradas da lista inteira e a segunda camada (c) os documentos restantes. Nesse cenário os valores de *blockMaxScore* obtidos são menores: 5, 6, 7, 15, e 18.

Considerando um exemplo de processamento de consultas, onde o atual limiar de descarte é 15 e o documento 13 é o pivô, ao processar usando o índice inteiro, esse pivô seria computado por estar em um bloco cujo *blockMaxScore* é 18, superior ao limiar de descarte. Enquanto no índice em camadas o documento pertence a um bloco cujo *blockMaxScore* é 5 garantindo o descarte do documento.

Apesar dessa estratégia permitir um número maior de saltos e descompressão mais úteis, há o custo de aumentar o número de termos da consulta. Durante o processamento cada nova camada é tratada como um termo diferente. Por exemplo, considerando índice separado em duas camadas uma dada consulta "A B" é processada como uma consulta com quatro termos.

Em [25] é apresentado o método BMW-CS. Desenvolvido com base nas técnicas de poda do método BMW, sendo um dos mais recentes e mais bem sucedidos de processamento de consultas em camadas. Seu funcionamento está baseado na divisão do índice em duas camadas, a primeira camada tem tamanho menor e é criada usando entradas que apresentam os maiores escores das listas invertidas, enquanto a segunda camada têm um índice muito maior e contém as entradas restantes. Essa divisão do índice é similar à

utilizada no método MBMW.

A ideia é usar a primeira camada para acelerar o processamento de consultas no segundo nível (ou seja, usando o índice maior) no cálculo do ranking final. O processamento é dividido em duas fases. A primeira fase, chamada de seleção de candidatos, processa apenas a primeira camada das listas dos termos da consulta. Nessa fase, o processo guarda uma lista com todos os documentos que tem chance de estar no ranking final, chamada lista de candidatos. Na segunda fase do processo, a segunda camada é percorrida apenas para buscar ocorrências dos documentos candidatos na segunda camada e computar o escore real desses documento.

Na primeira fase, a lista de candidatos é obtida percorrendo a primeira camada dos termos da consulta e computando a pontuação dos documentos encontrados. Uma vez que as listas invertidas não são completas (primeira camada), para computar o escore máximo de um pivô, o BMW-CS considera a hipótese desse documento ocorrer na segunda camada em todos os termos em que ele não aparece na primeira camada. Nesses casos, a pontuação máxima de um documento na segunda camada é adicionada para calcular sua pontuação máxima potencial, e se essa pontuação máxima potencial for maior do que o limite atual de descarte, esse documento será incluído na lista de candidatos.

A segunda fase percorre a segunda camada para calcular a pontuação total de cada documento candidato obtido durante o processamento da primeira camada. Documentos que não estão presentes na lista de candidatos são descartados, o que acelera consideravelmente o processamento na segunda camada. Após processar a segunda camada, o método garante que os escores corretos para cada documento candidato foram computados, obtendo assim um ranking dos *top-k* documentos candidatos.

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. Esse problema acontece porque documentos que não ocorrem em nenhuma das listas de termos da consulta na primeira camada não são avaliados. Nesse caso mesmo que o documento tenha escore suficiente para entrar na resposta, o algoritmo BMW-CS iria perdê-lo. Outra desvantagem desse método, apesar do bom desempenho, está no custo extra com memória para armazenar a lista de candidatos.

Capítulo 4

BMW-CSP e Waves

Neste capítulo, apresentamos a descrição detalhada dos algoritmos propostos para acelerar o processamento de consultas: BMW-CSP (*Block Max WAND with Candidate Selection and Preserving top-k results* e Waves.

O primeiro algoritmo, BMW-CSP, é uma evolução do método BMW-CS [25] que reduz seus requisitos de memória e evita as situações em que os documentos podem ser retirados de forma errada da resposta final. Um dos motivos para o desenvolvimento do método Waves foi reduzir o custo de armazenar a lista de candidatos durante o processamento com o método BMW-CSP e com o método BMW-CS. O método Waves também utiliza o índice dividido em camadas e aplica técnicas de poda semelhantes às apresentadas no método BMW [14]. O método Waves preserva a ordem dos documentos com melhor escore no resultado do processamento de consultas sem a necessidade de memória extra para a lista de candidatos.

4.1 BMW-CSP

O método BMW-CSP é uma evolução do método BMW-CS. Proposto em 2014, o método BMW-CS processa consultas no índice dividido em duas camadas usando técnicas de poda semelhantes às utilizadas no método BMW. No método BMW-CS o processamento de consultas é dividido em duas fases, uma fase para cada camada. A primeira fase consiste em processar a primeira camada da lista invertida de cada termo da consulta, onde estão os documentos mais relevantes. Nessa fase, o método busca documentos candidatos a resposta e os guarda em uma lista chamada lista de candidatos. A segunda fase

processa a segunda camada das listas invertidas em busca de ocorrências dos documentos candidatos, para computar o escore real de cada candidato. No fim dessa fase, o resultado para a consulta é gerado com os candidatos de maior peso. Com essa estratégia, o método consegue processar consultas de forma mais rápida que o método BMW. A desvantagem desse método é não garantir que a resposta para a consulta esteja ordenada de forma correta, pois não seleciona para a lista de candidatos os documentos que não ocorrem na primeira camada. Dessa forma, se um documento com escore suficiente para estar entre primeiras posições da resposta não aparecer na primeira camada em pelo menos um dos termos da consulta, esse documento será eliminado da resposta pelo BMW-CS, causando alteração no resultado que deveria ser mostrado.

A principal modificação proposta no método BMW-CSP foi desenvolvida para garantir a ordenação correta na apresentação de respostas. Para evitar as situações em que os documentos podem ser retirados de forma errada da resposta, o método BMW-CSP inclui uma terceira fase de processamento que é ativada no caso de não haver garantia de preservação dos resultados para cada consulta. Ou seja, a terceira fase é executada quando existir a chance de uma entrada que ocorre apenas na segunda camada dos termos da consulta ter peso suficiente para alterar a ordem dos resultados dentre os k documentos de maior escore. Tal situação ocorre quando a soma dos maiores escores dos termos da consulta na segunda camada do índice for maior do que o limiar de descarte após o processamento da segunda fase.

A terceira fase realiza uma segunda passagem na segunda camada para procurar novos documentos a serem incluídos entre os k resultados de maior escore. Os melhores resultados computados nas duas primeiras fases são utilizados para acelerar a terceira fase, uma vez que aumentam o limiar de descarte. Embora à primeira vista a execução de uma terceira fase pareça ser dispendiosa, nossos experimentos demonstram que ela não causa alterações significativas nos tempos finais de processamento de consulta quando se consideram os melhores cenários de configuração, uma vez que é raramente ativada. Por outro lado, com a terceira fase, o nosso algoritmo de processamento de consultas preserva os resultados, ao mesmo tempo que ainda se mantém mais rápido do que o algoritmo BMW.

Além da mudança para garantir a ordenação correta dos k resultados de maior escore, o BMW-CS também foi modificado para reduzir os requisitos de memória e tempo de processamento das consultas. As três mudanças principais incluem:

- Calcular um limiar inicial de descarte para a fase de seleção de candidatos, como apresentado por Carvalho et al. [12]. Enquanto o limiar inicial do BMW-CS é zero, nosso método começa com um limiar inicial mais elevado e seguro que descarta mais documentos no início do processamento da consulta. O limiar inicial é definido como o maior k-ésimo escore entre os termos da consulta, uma vez que este é um limite inferior claro para a menor pontuação entre os documentos que podem estar entre os *k* resultados de maior escore;

- Usar o *block max score* a partir do índice da segunda camada ao selecionar candidatos na primeira etapa. Com essa mudança, obtemos uma melhor estimativa do *max score* de cada documento candidato na primeira fase, reduzindo assim a possibilidade de incluir candidatos que não vão estar presentes nos resultados finais. No BMW-CS, tais estimativas foram baseadas na pontuação máxima do termo na segunda camada, em vez de usar a pontuação do *block max score*. Essa mudança não só acelerou o processamento das consultas, mas também reduziu o número de candidatos, e portanto a sobrecarga de memória requerida pelo método;

- Aproveitar o limiar de descarte, já computado ao processar a primeira camada, como limiar inicial para ser usado no cálculo do escore completo, na segunda fase. Esta estratégia simples não foi usada pelos autores do BMW-CS original, o que permitiu outra vez acelerar o processamento da consulta.

Durante a seleção dos candidatos, a estrutura de um *heap* de mínimo é usada para armazenar os *k* documentos de maior escore. A pontuação mais baixa do *heap* é usada como o limiar de descarte para podar dinamicamente entradas sem chance de fazer parte do resultado final. Todos os documentos analisados que têm uma pontuação máxima maior do que o limiar de descarte são adicionados ao conjunto de candidatos. Em seguida, a segunda camada é processada para calcular os escores completos de todos os documentos candidatos. Depois de calcular o escore real, o algoritmo verifica se o resultado fornecido pela segunda fase preserva os *k* resultados de maior escore da resposta. Isso é verificado pelo cálculo da pontuação máxima que um documento com entradas apenas na segunda camada iria conseguir. Esse cálculo é a soma de todas as pontuações máximas de termos na segunda camada. Essa soma é então comparada com o limiar de poda atual. Se a soma for maior do que o limiar de poda, isto significa que existe uma possibilidade de um do-

cumento com entradas só na segunda camada estar presente na resposta final. Nesse caso, a terceira fase é necessária.

Na seção a seguir, apresentamos a implementação em maior detalhe para tornar a reprodução do método mais fácil para o leitor.

4.1.1 Detalhes de implementação

A Listagem 4.1 apresenta uma visão geral do algoritmo BMW-CSP, ilustrando ainda mais o seu funcionamento. Em sua primeira fase, BMW-CSP usa o primeiro nível para selecionar documentos que são candidatos a compor o topo dos resultados. Inicialmente, o conjunto de documentos candidatos, denotado por $\mathcal{A}$, é gerado pela função *SelectCandidates*, que calcula os candidatos para estarem entre os k resultados de maior escore da resposta.

Listing 4.1: Algoritmo BMW-CSP

```
1  BMW_CSP(queryTerms[1..q], k)
2   I_secondTier é a segunda camada do índice
3   θ é o limiar de descarte
4   H é o heap de mínima para manter os documentos mais relevantes
5   A é a lista de candidatos
6
7   lists ← I_secondTier(queryTerms); //Seleciona a segunda camada das listas
8
9   // Phase 1: Seleção de candidatos
10  A ← SelectCandidates(queryTerms, k, θ);
11
12  // Phase 2: Calcula escore completo, adota o θ da primeira fase
13  H ← CalculateCompleteScore(A, queryTerms, k,θ);
14
15  // Phase 3: Garante a integridade dos resultados
16  if(H_1.score < Σ_{i=1}^{|lists|} lists[i].maxScore2Tier)
17    H ← guaranteeTopKResults(A, H, queryTerms, k);
18  end if
19  sort H by score;
20
21  return H;
```

Seleção de candidatos

A Listagem 4.2 descreve o algoritmo de seleção dos candidatos, o que corresponde à fase 1 do nosso método. Ele é iniciado selecionando as listas invertidas a serem processadas (linha 6), que são as listas que representam cada termo de consulta. O limiar de des-

carte θ é inicialmente definido como o valor máximo entre os k-ésimos maiores escores encontrados em cada lista (linha 7).

A linha 8 faz cada uma das listas invertidas apontar para o seu primeiro documento. A função $Next(l,d)$ busca na *skiplist* associada com a lista l o bloco onde há a primeira ocorrência de um $docId$ igual ou maior do que d, fixando o bloco atual para a posição encontrada. Em seguida, ele move o ponteiro para o documento atual da lista, $l.curDoc$, para a menor entrada com valor superior a d.

As listas invertidas dos termos da consulta exibidos na Listagem 4.2 são representadas pelo vetor $lists$ e cada lista invertida tem um ponteiro interno para um docId que esta sendo processado no momento, chamado de $docId$ corrente. Na linha 10, o vetor é ordenado em ordem crescente de acordo com o $docId$ corrente de cada uma dessas listas. Em seguida, na linha 11, é computado o próximo documento que tem chance de estar presente no topo dos resultados, executando o *pivoting()*. O nosso pivô é calculado levando em consideração a possibilidade de algumas das entradas do documento não estarem incluídas na primeira camada das listas, o que nos permite adicionar informação para calcular um escore aproximado (*upScore*), que é uma pontuação máxima possível, de cada documento selecionado como pivô. Esse procedimento é descrito na Listagem 4.3. Linhas 12 e 13 testam a condição de parada e definem o documento atual a ser analisado pelo algoritmo.

A Linha 14 usa a função *NextShallow* para mover os ponteiros para o bloco do documento corrente em cada lista. A função *NextShallow* é a mesma apresentada na proposta original do BMW, e difere da função *Next* porque ela não precisa acessar os documentos, mas apenas as *skiplists* das listas invertidas para mover seus ponteiros e definir um novo bloco corrente (bloco onde é provável que o pivô ocorra). Usando esta função, podemos ignorar entradas da lista sem a necessidade de acessá-las. A Linha 15 chama a função *CheckBlockMax*, detalhada na Listagem 4.4.

O restante do algoritmo checa se um documento tem uma pontuação alta o suficiente para ser incluído na resposta. A Linha 19 usa as pontuações máximas dos termos no segundo nível para dar uma estimativa inicial de *upScore*. O escore de cada documento é usado na condição de incluí-lo no *heap* $\mathcal{H}$, como nas Linhas 20 a 25. $\mathcal{H}$ é mantido para controlar o limite de descarte θ. O escore máximo de cada documento (*upScore*) é utilizado na Linha 26 para verificar se um documento deve ser incluído no conjunto de candidatos $\mathcal{A}$. Se a primeira estimativa do *upScore* é alta o suficiente para incluir

Listing 4.2: Algoritmo *SelectCandidates*

```
1  SelectCandidates (queryTerms[1..q], k, θ)
2    H é o heap de mínima para manter os documentos mais relevantes
3    A é a lista de candidatos
4    I_cand é a primeira camada
5
6    lists ← I_cand(queryTerms); //Carrega lista invertida da primeira camada
7    θ ← MAX_{i=1}^{|lists|} lists[i].k–thScore;
8    for i ← 1 to |lists| do Next(lists[i], 0); //Aponta para o primeiro docid das listas
9    repeat
10     sortByCurrentPointedDocId(lists);
11     p ← Pivoting(lists, θ);
12     if (p == −1) break; //Sem mais candidatos
13     d ← lists[p].curDoc;
14     for i ← 1 to p do NextShallow(lists[i], d); //Move os ponteiros da skiplist
15     if (CheckBlockMax(lists, θ, p) == TRUE)
16       if (lists[1].curDoc == d)
17           doc.docId ← d;
18           doc.score ← Σ_{i=1}^{p} BM25(lists[i]);
19           doc.upScore ← doc.score + Σ_{i=p+1}^{|lists|} lists[i].maxScore2Tier;
20           if (|H| < k) H ← H ∪ doc;
21           else if (H_1.score < doc.score)
22                    H ← H − H_1; //Remove a entrada com menor escore
23                    H ← H ∪ doc;
24                    θ ← H_1.score; //Atualiza o limiar
25                end if
26           if (θ <= doc.upScore)
27              doc.upScore ← doc.score + Σ_{i=p+1}^{|lists|} lists[i].blockMaxScore2Tier(d);
28              if (θ <= doc.upScore)
29                  doc.terms ← queryTerms[1..p];
30                  A ← A ∪ doc;
31                  if ({∃ dLow ∈ A| dLow.upScore < θ}) A ← A− dLow;
32              end if
33           end if
34           for i ← 1 to p do Next(lists[i], d+1); // Move listas avaliadas
35       else
36           j ← {x|lists[x].curDoc < d ^ |lists[x]| < |lists[y]|,∀1 ≤ y < p};
37           Next(lists[j], d);
38       end if
39     else
40       minDoc ← MAXDOC; //Obtém o menor docId entre os atuais block boundaries das listas
41       for i ← 1 to p do
42         firstNext ← lists[i].getDocBlockBoundary(); //Primeiro docId do próximo bloco
43         if(minDoc > firstNext) minDoc ← firstNext;
44       end for
45       j ← {x|1 ≤ x ≤ p ^ |lists[x]| < |lists[y]|,∀1 ≤ y ≤ p};
46       Next(lists[j], minDoc);
47     end if
48   end repeat
49   for each {A_i ∈ A|A_i.upScore< θ} do A ← A − A_i;
50   return A;
```

Listing 4.3: Algoritmo *Pivoting*

```
1  Pivoting (lists, θ)
2    accum ← 0;
3    for i ← 1 to |lists| do
4      accum ← accum + lists[i].maxScore;
5      accumMin ← Σ_{j=i+1}^{|lists|} lists[j].maxScore2Tier;
6      if (accum + accumMin >= θ)
7        while(i+1<=|lists| AND lists[i+1].curDoc == lists[i].curDoc) do
8            i ← i + 1;
9        end while
10       return i;
11     end if
12   end for
13   return −1;
```

Listing 4.4: Algoritmo *CheckBlockMax*

```
1 CheckBlockMax (lists, p, θ)
2
3  //Soma a escore máximo dos blocos onde o documento atual pode aparecer
4  max ← ∑_{i=1}^{p} lists[i].getBlockMaxScore();
5
6  //Adiciona o escore máximo dos demais termos na segunda camada
7  max ← max + ∑_{i=p+1}^{|lists|} lists[i].maxScore2Tier;
8  if(max > θ) return true;
9  return false
```

o documento como um candidato, uma estimativa mais apertada é calculada usando o *blocoMaxScore* da segunda camada, como mostrado na Linha 27.

O θ muda quando mais documentos são computados, por isso, sempre que um documento é adicionado em $\mathcal{A}$, também é verificado se existe pelo menos um documento em $\mathcal{A}$ com *upScore* menor do que o valor atual de θ. Em tais casos, o documento é removido de $\mathcal{A}$ (Linha 31). Esse procedimento evita o desperdício de memória em manter elementos em $\mathcal{A}$ que serão descartados no fim do processo. Depois de atualizar $\mathcal{A}$, o algoritmo move os ponteiros de cada uma das listas invertidas para continuar o processamento da consulta. Esse movimento é realizado da Linha 34 à 46.

Por fim, o algoritmo poda o conjunto $\mathcal{A}$, removendo todos os candidatos que não têm chance de estar presentes entre os k resultados de maior escore (Linha 48). Essa poda diminui o custo da segunda fase. O final do algoritmo de seleção de candidatos resulta na lista de documentos candidatos $\mathcal{A}$, de modo que o resultado final pode ser obtido processando o restante do índice na segunda fase.

Computação escore completo

A função *CalculateCompleteScore* (Listagem 4.5) implementa a fase 2 do algoritmo. Nessa função, os escores dos candidatos com os termos ausentes são computados utilizando o índice maior, que contém as entradas que não estão presentes na primeira camada. Nessa etapa, o valor inicial do limiar θ é definido como o mesmo obtido no final da fase de seleção de candidatos. Esse procedimento permite descartar mais entradas e acelerar a segunda fase. Além disso, para evitar custos desnecessários de descompressão ou acessar as entradas da lista, o movimento superficial descrito em [14] é usado para alinhar todas as listas. Em seguida, uma segunda verificação do *BlockMaxScore* é feita para verificar se o documento ainda tem chance de estar entre os k resultados de maior escore. Cada docu-

mento é avaliado somente se tiver uma pontuação alta o suficiente, caso contrário, ele será descartado. Como na primeira fase, mantemos um *heap* de mínimo, com os documentos com os maiores escores avaliados, e a pontuação mínima desse conjunto atualiza o θ.

Listing 4.5: Algoritmo *CalculateCompleteScore*

```
1  CalculateCompleteScore( A, queryTerms[1..q], k,θ)
2    H é o heap de mínima para manter os documentos mais relevantes
3    I_second_tier é a segunda camada do índice
4    lists ← I_second_tier(queryTerms);
5    H ← ∅;
6    sort A by docId;
7    for i ← 1 to |A| do
8        if (A_i.score < A_i.upper_score)
9           localUpperScore ← A_i.score;
10          for j ← 1 to |lists| do
11              if (queryTerm[j] ∉ A_i.terms)
12                 NextShallow(lists[j], A_i.docId);
13                 localUpperScore ← localUpperScore + lists[j].getBlockMaxScore();
14             end if
15          end for
16          if (localUpperScore > θ)
17             for j ← 1 to |lists| do
18                 if (queryTerm[j] ∉ A_i.terms) Next(lists[j], A_i.docId);
19             end for
20             for each {1 ≤ x ≤ |lists| | lists[x].curDoc = A_i.docId} do
21                 A_i.score ← A_i.score + BM25(lists[x]);
22             end for
23          end if
24       end if
25       if (θ < A_i.score)
26          if (|H| < k) H ← H ∪ A_i;
27          else if (H_1.score < A_i.score)
28                H ← H − H_1; // Remove a entrada com menor escore
29                H ← H ∪ A_i;
30                θ ← H_1.score;
31             end if
32       end if
33   end for
34   return H;
35 end
```

Preservando a Ordenação de Resultados

Para garantir a ordenação correta dos k resultados de maior escore da resposta, a terceira fase realiza uma segunda passagem na segunda camada. Na Listagem 4.1, essa fase é representada pela função *guaranteeTopKResults*, que atualiza, quando necessário, a lista

dos k documentos armazenados no *heap* $\mathcal{H}$ com os documentos cujas entradas ocorrem somente na segunda camada. Essa função é muito semelhante a uma execução do método BMW na segunda camada. As únicas diferenças são: (i) ele começa com o *Heap* $\mathcal{H}$ contendo os k documentos de maior escore computados até esse ponto; e (ii) os documentos que já foram computados não são considerados nessa fase, para evitar a repetição de resultados. Observe que, como os escores na segunda camada são menores do que os da primeira camada, espera-se que a maioria das entradas sejam descartadas e, portanto, essa terceira fase deverá tornar-se mais rápida do que a execução regular do BMW.

4.2 WAVES

A principal motivação para o desenvolvimento do método Waves foi reduzir a necessidade de espaço extra para armazenar os dados parciais necessários durante o processamento de consultas com o método BMW-CSP.

Além de não necessitar do espaço extra para armazenar documentos candidatos à resposta, o método Waves obteve o melhor desempenho em tempo de processamento quando comparado aos outros métodos que avaliamos. O método Waves processa as consultas através da realização de várias passadas nas camadas do índice, que chamamos de *waves*. Cada *wave* começa a partir de uma camada i e calcula somente a pontuação dos documentos que ocorrem em i que têm a chance de estar presentes no resultado final. Para isso, o método acessa as entradas do documento não só na camada i, mas também em todos os níveis $j > i$, assim, cada *wave* atravessa o índice da camada atual passando por todas as camadas restantes.

Sempre que a *wave* de uma camada i termina, o algoritmo checa se existe uma garantia de que os k resultados de maior escore já foram computados ou não. Se não houver essa garantia, o algoritmo executa uma nova *wave* a partir da camada $i+1$ em busca de novos documentos que possam entrar no topo da resposta final dada para a consulta.

A cada momento durante o processamento de uma consulta, são mantidos os k documentos com os mais altos escores encontrados até aquele momento. A pontuação mínima deste conjunto é usada como um limiar de descarte para verificar se um novo documento tem chance de fazer parte do resultado final ou não. Ao iniciar uma nova *wave*, os k resultados de maior escore encontrados por *waves* anteriores aceleraram ainda mais a travessia,

Listing 4.6: Algoritmo Waves

```
1  Waves (Q,k, I[1..m])
2   Q é o conjunto de termos da consulta
3   I[1..m] é o índice dividido em m camadas
4   k é o número de elementos que devem ser retornados
5   H é o heap para armazenar os k resultados de maior escore
6   maxS(t,j) é o escore máximo do termo t na camada j
7   MAX(S) é o valor máximo entre os elementos do conjunto S
8
9   H ← ∅;
10  θ ← MAX({maxS(t,2)|t ∈ Q});
11  for i = 1 to m do
12      H ← waveExecution(Q, H,i,I[1..m],θ)
13      θ ← H_0.score;
14      if((i < m) and (θ ≥ ∑_{∀t∈Q} maxS(t,i+1)))
15          break;
16      end if
17  end for
18  sort H by score;
19  return H;
```

descartando mais documentos.

Uma nova *wave* começa a partir de uma camada i sempre que a soma das pontuações máximas dos termos da consulta no nível i seja superior ao limiar de descarte, uma vez que, nesses casos, pode haver documentos com chance de estar no resultado final mas que ainda não foram incluídos na lista dos k resultados de maior escore.

Embora à primeira vista a execução de várias passadas nas camadas do índice pareça um processo dispendioso, nossos experimentos mostram que, de fato, o método Waves reduz o tempo médio de processamento de consultas, uma vez que o número de *waves* necessárias para processar a maioria das consultas tende a ser pequeno.

A Listagem 4.6 apresenta uma visão geral do nosso algoritmo. Cada *wave* atravessa da camada atual até o último nível, mas são avaliados apenas os documentos que aparecem pelo menos uma vez na camada atual. O método *Waves* começa calculando um limiar inicial de descarte θ, como proposto para o método BMW-CSP.

Depois de definir o limiar inicial, o algoritmo executa as *waves*, fazendo sucessivas chamadas para a função *waveExecution* (linha 12). Dada uma camada i, a função *waveExecution* verifica quais documentos, dentre os que ocorrem pelo menos uma vez em i, devem ser incluídos entre as atuais respostas ($\mathcal{H}$). Nós detalhamos essa função na Listagem 4.7.

Depois de terminar a execução de uma *wave*, o algoritmo verifica se qualquer documento ainda não avaliado poderia eventualmente fazer parte dos k documentos de maior escore. A verificação é executada nas Linhas 14-16 do algoritmo. Se a soma da pontuação

Listing 4.7: Algoritmo *WaveExecution*

```
1  WaveExecution (Q, H, i, I[1..m], θ)
2   Q é o conjunto de termos da consulta
3   H é o heap para armazenar os k resultados de maior escore
4   i é a camada atual
5   I[1..m] é o índice dividido em m camadas
6   θ é o limiar de descarte atual
7   D(t) é o docId apontado na camada i do índice I para um dado termo t
8   maxS(t,j) é o escore máximo do termo t na camada j
9   maxBl(d,t,j) é o blockMaxScore de d no termo t na camada j
10  MIN(S) é o menor id do conjunto S
11  move list pointers to the beginning of tier i ;
12  while (have documents in the tier to inspect)
13     pivot ← MIN({D(t)|t ∈ Q ∧ Σ_{∀t'∈Q|D(t')≤D(t)} maxS(t',i) + Σ_{∀t'∈Q|D(t')>D(t)} maxS(t',i+1) >θ});
14
15     estimate ← (Σ_{∀t'∈Q|D(t')≤pivot} maxBl(pivot,t',i) + Σ_{∀t'∈Q|D(t')>pivot} maxS(t',i+1));
16     if (estimate > θ)
17        estimate ← (Σ_{∀t'∈Q|D(t')≤pivot} maxBl(pivot,t',i) + Σ_{∀t'∈Q|D(t')>pivot} maxBl(pivot,t',i+1));
18        if (estimate > θ)
19              score ← compute the score of pivot taking their frequencies from the index;
20              if( score > θ)
21                    update H with the pivot and its score;
22                    θ ← H_0.score; // menor escore em H
23              endif
24        endif
25     endif
26     move lists pointers forward;
27  end while
28  return H;
```

máxima possível (ou *maxScore*) no próximo nível dos termos da consulta for maior do que o limiar de descarte atual, então há uma chance de incluir novos documentos na resposta, e uma nova *wave* é iniciada. Caso contrário, o algoritmo termina e os k resultados finais computados, os de maior escore, são ordenados para produzir a resposta final.

A Listagem 4.7 apresenta detalhes sobre como processamos cada *wave* em nosso algoritmo. Durante o processamento de uma *wave*, o algoritmo mantém um ponteiro para a posição atual de cada uma das listas invertidas dos termos da consulta na camada i atual. O algoritmo começa movendo os ponteiros de cada lista para o primeiro documento de cada lista (Linha 11). Em seguida, ele repete a etapa de seleção de documentos até que os ponteiros cheguem ao final da camada atual.

A seleção de documentos é realizada utilizando um passo de avaliação múltipla. Em primeiro lugar, usamos a pontuação máxima possível (*maxScore*) de documentos na camada atual e na próxima camada. Nós selecionamos como pivô o menor *docID* atualmente apontado cuja soma de suas pontuações máximas é maior do que o limiar de descarte θ (Linha 13). A pontuação máxima de um documento d é a soma de suas pontuações máximas em todos os termos da consulta. Para os termos em que o documento apontado é menor ou igual a d, há uma chance de encontrar o documento d, nesse caso, a pontuação

máxima é retirada do nível i. Para os demais termos, sabemos que o documento d não ocorre na camada i, e, portanto, tomamos a pontuação máxima da camada $i+1$. Dado um índice com M camadas, vamos atribuir valores de pontuação máxima como zero após a camada M.

Para o pivô selecionado na primeira etapa de avaliação, calculamos uma estimativa de escore mais aproximada (linha 15), utilizando o *Block Max Score* do pivô nas listas em que o documento pode ocorrer na camada i. Novamente, para os termos que não contêm o pivô na camada i, tomamos a pontuação máxima na camada $i+1$, considerando o *Block Max Score*. É uma estimativa de escore máximo mais aproximada do escore real do documento, mas que ainda considera a possibilidade de todos os termos da consulta ocorrerem no documento.

Se um candidato é aprovado em todos os passos anteriores, é considerado um documento com potencial para ser incluído na resposta. Realizamos uma avaliação final para computar seu escore. Esse procedimento requer acesso às listas invertidas para a leitura da frequência de cada termo da consulta no documento pivô. A leitura da frequências implica na descompressão de um bloco de documentos da lista invertida de cada termo da consulta, uma vez que os índices invertidos são normalmente armazenados em um formato compactado. A informação de frequência é retirada da lista na camada i se o pivô ocorre na mesma. Para obter informações dos termos que não contém o documento na camada i, as camadas seguintes são verificadas até encontrar o pivô ou até a última camada ser inspecionada. Note que, na pior das hipóteses, podemos acessar um bloco por camada para ler a frequência de um determinado termo no documento. Depois de computar o escore do documento pivô, o algoritmo atualiza o *heap* se o escore superar o atual limiar de descarte (Linhas 20-23).

Concluindo o processo de avaliação de um documento, o algoritmo avança os ponteiros das listas invertidas (Linha 26) e repete o processo de avaliação para o novo conjunto de documentos apontados. Informações sobre o final dos blocos podem ser usadas para fazer movimentos mais agressivos em casos especiais, usando ideias semelhantes às descritas por Ding e Suel [14].

Quando o pivô é totalmente avaliado, os ponteiros são apenas movidos para a próxima entrada maior do que o pivô em cada lista. Para as demais situações, os ponteiros são movidos para a próxima entrada de d, de tal forma que:

$$d \leftarrow MIN(\{Boundary(pivot,t,i)|D(t) \leq pivot \wedge t \in Q\} \cup \{D(t)|D(t) > pivot \wedge t \in Q\}) \quad (4.1)$$

onde $Boundary(x,t,i)$ é uma função que recebe o ID do primeiro documento após o bloco onde documento x ocorre na camada i do termo t e MIN, Q, D(t) e pivot são definidos como na Listagem 4.7.

4.2.1 Exemplo

Nesta seção vamos usar a Figura 4.1 para exemplificar o processamento de consultas do algoritmo *Waves*. No exemplo da figura, o índice é dividido em duas camadas. Vamos considerar que a atual *wave* está processando a primeira camada do índice e os documentos apontados nas listas são 69, 466 e 900, respectivamente, para os termos da consulta 'Very', 'Large' e 'Data'. Nesse exemplo, o documento 69 ocorre na lista do termo 'Very' na primeira camada. Para verificar se ele tem potencial para se tornar um dos k resultados de maior escore, o algoritmo computa seu escore máximo, $1,8$ mais $1,7$. O valor de $1,7$ vem da soma de $1,1+0,6$, e representa a pontuação máxima que receberia na hipótese dele ocorrer na segunda camada dos termos 'Large' e 'Data'. Observe que as listas de 'Very' e 'Large' estão apontando para *docIds* superiores a 69, significando que o documento 69 não ocorre na camada 1 para essas listas. Assim, tomamos a pontuação máxima na segunda camada.

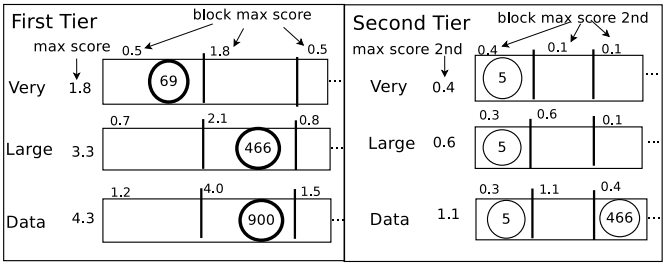

Figura 4.1: Lista invertida ao processar a consulta 'Very Large Data' com índice em duas camadas.

A pontuação máxima potencial para o documento 69 na coleção é 3,5. Se essa pontuação for maior do que o limiar de descarte θ, *Waves* acessa a primeira camada para obter o *block Max Score* do documento 69 em cada lista que o documento pode ocorrer.

O segundo nível de avaliação, resulta em $0,5+1,1+0,6=2,2$. Se esse limite superior mais baixo também supera o limite de θ, *Waves* calcula uma terceira estimativa de escore para o documento 69, usando o *block Max Score* dos termos 'Large' e 'Data' na segunda camada, obtendo uma estimativa de escore ainda mais baixa. Se esta terceira estimativa ainda for superior a θ, o índice é acessado para computar o escore real do documento.

Ao processar uma camada *i*, documentos que ocorrem apenas nas próximas camadas das listas não são avaliados. Em alguns casos esses documentos que não ocorrem na camada *i* estão entre os *k* documentos de maior escore para a consulta e portanto devem ser avaliados. Essa situação só pode ocorrer quando a soma das pontuações máximas dos termos da consulta no nível $i+1$ for maior do que o limiar de descarte após o processamento da consulta na camada *i*.

No exemplo da Figura 4.1, a soma das pontuações máximas dos termos da consulta na segunda camada é $2,1$. Se o limite de descarte θ após o processamento da primeira *wave* for menor do que $2,1$, a próxima *wave* é iniciada atravessando a segunda camada para garantir que o resultado final estará correto. No exemplo, o segundo nível é a última camada. Depois de processá-lo os *k* resultados de maior escore estarão devidamente calculados. Com esse passo, os documentos que ocorrem apenas na segunda camada, como o documento 5, são avaliados para garantir que o resultado da consulta será preservado.

Uma vez que os resultados apurados nas *waves* anteriores são usados na atual, e dado que os pesos dos documentos na camada atual são mais baixos do que os pertencentes às camadas anteriores, cada nova *wave* representa uma sobrecarga de tempo pouco significativa.

Capítulo 5

Experimentos

Neste capítulo apresentamos os experimentos realizados para avaliar os métodos de processamento de consultas considerados neste trabalho. Inicialmente, apresentamos detalhes sobre a configuração dos experimentos como: (i) coleção de documentos, (ii) coleção de consultas, (iii) função de similaridade e (iv) objetivos definidos para a realização dos experimentos. Em seguida, apresentamos uma descrição dos experimentos e os resultados alcançados com cada método de processamento de consultas avaliado.

5.1 Configurações

Para a avaliação dos métodos de processamento de consultas adotamos a coleção de referência *TREC GOV2* na maior parte dos experimentos e realizamos experimentos complementares com uma segunda coleção, a ClueWeb09, com 50 milhões de documentos. *TREC GOV2* é composta por 25.205.179 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 vocabulário é composto por cerca de 40 milhões de termos distintos. A coleção TREC GOV2 foi adotada por ter se tornado um padrão para estudos que propõem melhorias de eficiência em sistemas de busca, tendo sido usada nos experimentos dos principais trabalhos encontrados na literatura.

Selecionamos dois sub-grupos de consultas da coleção *TREC 2006 efficiency queries*. O primeiro conjunto contém 1.000 consultas e é usado em experimentos iniciais realizados para definir a melhor divisão do índice em camadas para os métodos multi-camadas avaliados, e será referido como *conjunto de calibração*. O segundo conjunto contém

10.000 consultas e é usado em experimentos para comparar o desempenho dos métodos com os melhores parâmetros escolhidos com o conjunto de calibração, e será referido como *conjunto de teste*.

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 anteriores [25, 14, 32] incluindo nossos *baselines*, 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.

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 outro método de *ranking* 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 usando como métricas: o tempo de resposta, o número de acessos às listas invertidas e a quantidade de documentos computados. Junto ao tempo médio de processamento, é apresentado nas tabelas desse capítulo o intervalo de confiança ($\pm$) de 95% para o tempo médio de processamento das consultas, indicando que o tempo médio estará dentro do intervalo apresentado em 95% das amostras. Para verificar o impacto do tamanho da coleção no desempenho dos métodos experimentados, apresentamos experimentos com diferentes tamanhos da coleção. Além da coleção completa, que chamamos de GOV2, nós também realizamos experimentos com variações contendo 1 milhão de documentos, que chamamos de GOV2-1, e contendo 10 milhões de documentos, que chamamos de GOV2-10.

Implementamos os *baselines*: WAND [8], BMW e MBMW [14]. Os códigos foram desenvolvidos em C++ tendo como base trabalhos anteriores. Adotamos o modelo probabilístico BM25 como função de similaridade. Os códigos foram comparados com os códigos originais do BMW para evitar diferenças na implementação que poderiam afetar o desempenho. O MBMW [14] foi escolhido como *baseline* porque os seus autores mostram que o método supera as principais alternativas de *baseline* que encontramos

na literatura, incluindo o TAT mais eficiente e algoritmos DAT. Nós também realizamos experimentos com o *Hybrid*, um outro método de processamento de consultas que não havia sido comparado com o BMW. *Hybrid* requer espaço adicional para guardar os resultados parciais (acumuladores) e alcançou o pior desempenho em nossos experimentos, portanto, nós não relatamos seus resultados para não acrescentar ruído à comparação com os melhores *baselines* que encontramos.

Os principais objetivos com os experimentos são: (i) Estudar o impacto do tamanho da coleção em cada método. O tamanho da coleção pode afetar a escolha do melhor método ou melhor divisão do índice em camadas. Por exemplo, listas invertidas menores têm menos blocos, e assim as heurísticas para descartar blocos podem não funcionar tão bem.(ii) Verificar se o melhor parâmetro para divisão do índice em camadas, selecionado para o conjunto de consultas de calibração, produz resultados competitivos ao processar o conjunto de consultas de teste. A seleção do tamanho de cada camada e número de camadas é um passo importante quando se utiliza índices em múltiplas camadas, uma vez que esses parâmetros podem afetar o desempenho final dos métodos. (iii) Verificar se as novas estratégias de processamento de consultas são eficazes. Queremos mostrar na prática que cada wave amortiza os custos para a próxima wave, uma vez que aumenta o limiar de descarte, reduzindo o número de elementos verificados nas próximas waves. Além disso, o custo de cada wave em si é amortizado pela restrição de avaliar apenas os documentos que ocorrem na camada atual e que têm potencial para ultrapassar o limiar de descarte. O algoritmo pode ainda descartar waves, o que pode reduzir ainda mais os seus custos.

Na primeira fase de experimentos, Seção 5.2, todas as *stopwords* foram removidas tanto do menor conjunto de consultas quanto do maior conjunto. A melhor divisão do índice é definida para cada método multi-camada com o conjunto de consultas de calibração, em seguida, na Seção 5.3 a melhor configuração definida para cada método é reavaliada com o conjunto de consultas de teste. Na Seção 5.4, os métodos são avaliados sem remover as *stopwords* das consultas por esse ser um cenário mais realista. Na Seção 5.5, são apresentados resultados de experimentos com uma técnica de redistribuição de identificadores de documentos [30] que atribui identificadores semelhantes aos documentos semelhantes durante a indexação. Essa técnica ajuda a melhorar a desempenho dos métodos.

Para reforçar os experimentos e avaliar o comportamento dos métodos em uma coleção ainda maior, na Seção5.6 são apresentados experimentos com um subconjunto de documentos da coleção ClueWeb09, o subconjunto da categoria B (Inglês 1), que consiste em cerca de 50 milhões de páginas da Web em Inglês. Consideramos as primeiras 1000 consultas da TREC 2009 Million Query 40k (MQ09#40K) e adotamos as mesmas distribuições ideais de tamanhos de camadas para cada método definidas com a GOV2. A coleção foi ordenada por URLs para determinar os identificadores dos documentos e as *stopwords* foram mantidas nas consultas.

5.2 Melhor divisão em camadas

O tamanho de cada camada do índice é crucial para o desempenho dos métodos que usam índice invertido em camadas. Nesta seção, apresentamos o estudo desenvolvido para identificar a melhor divisão em camadas para cada um dos métodos avaliados. Com isso, garantimos uma comparação justa, considerando apenas o melhor desempenho de cada método.

Adotamos a mesma estratégia de seleção global proposta em [25] para selecionar as entradas para cada camada no processo de divisão do índice. Na abordagem global, estima-se um único limiar de corte que seja capaz de dividir o índice em duas camadas. Esse limiar é baseado na distribuição de valores de escore de todas as entradas do índice.

Calculamos os limiares de divisão adaptando os valores para os tamanhos desejados em cada camada. Para a primeira camada, nós selecionamos as entradas com escores acima do limiar de divisão, adotando um tamanho mínimo de 1000 entradas para evitar que a primeira camada de qualquer lista fique muito pequena. Assim, as listas com tamanhos menores do que 1000 serão totalmente armazenadas na primeira camada. As entradas menores que o limiar global são armazenadas nas camadas restantes.

Para selecionar a melhor divisão em camadas utilizamos a coleção de calibração, que possui mil consultas. Na primeira fase dos experimentos dividimos o índice em duas camadas. Foram geradas 25 versões do índice com a primeira camada Δ variando de 2 até 50% do índice na primeira camada à razão de 2%, como pode ser visto nos gráficos da Figura 5.1, onde também são apresentados os resultados dessa primeira fase de experimentos.

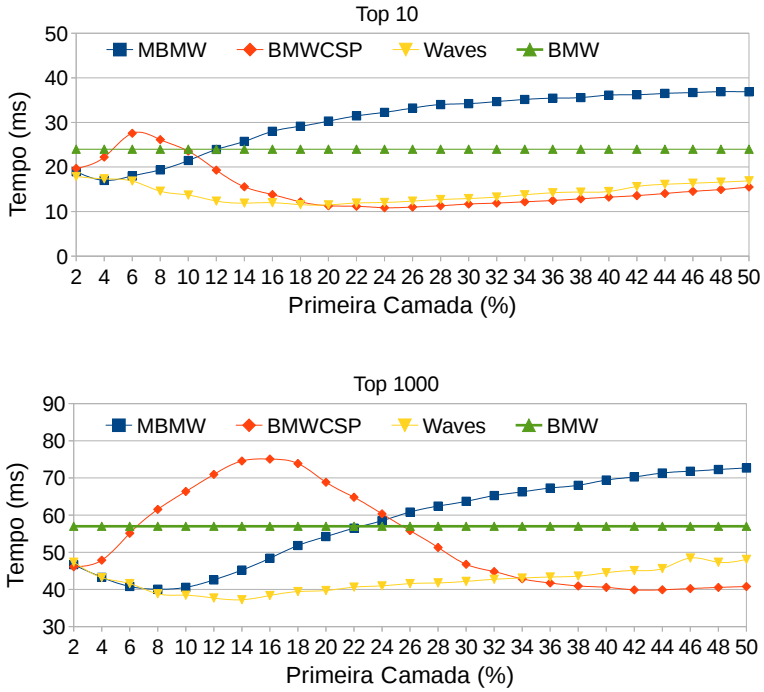

Figura 5.1: Duas camadas: Variação do tempo de processamento dos métodos para diferentes divisões do índice em camadas. Processando top-10 e top-1000 respostas.

Foi acrescentado ao gráfico da Figura 5.1 o tempo de processamento do método BMW que usa o índice inteiro, apenas como uma referência. Com exceção do método BMW cada ponto nos gráficos representa o tempo de processamento de um método para uma dada divisão do índice em duas camadas. Cada método tem seu melhor desempenho em um ponto de divisão do índice diferente dos demais. Ao computar os top-1000 resultados os melhores tempos são $40,0$ *ms*, $39,9$ *ms* e $37,2$ *ms*, respectivamente para os métodos MBMW com $\Delta = 8\%$, BMW-CSP com $\Delta = 42\%$ e WAVES com $\Delta = 14\%$. Ao computar os top-10 resultados, os melhores tempos são $17,0$ *ms*, $10,9$ *ms* e $11,5$ *ms* para os métodos MBMW com $\Delta = 4\%$, BMW-CSP com $\Delta = 24\%$ e WAVES com $\Delta = 20\%$.

Os resultados apresentados na Figura 5.1 correspondem ao desempenho dos métodos com o índice em duas camadas. O método BMW-CSP como variante do método BMW-CS tem como proposta trabalhar com índice em duas camadas, já os métodos Wave e MBMW podem trabalhar com N camadas. Na próxima etapa dessa seção, avaliamos o desempenho dos métodos ao processar consultas com variações do índice em três camadas. Para esses experimentos foram geradas 42 variações do índice em três camadas. Para cada primeira camada com Δ igual a 1, 3, 5, 15 e 20% foram geradas variações da segunda camada com Δ igual a 5, 10, 15, 20, 25, 30 e 35%. Os tempos de processamento

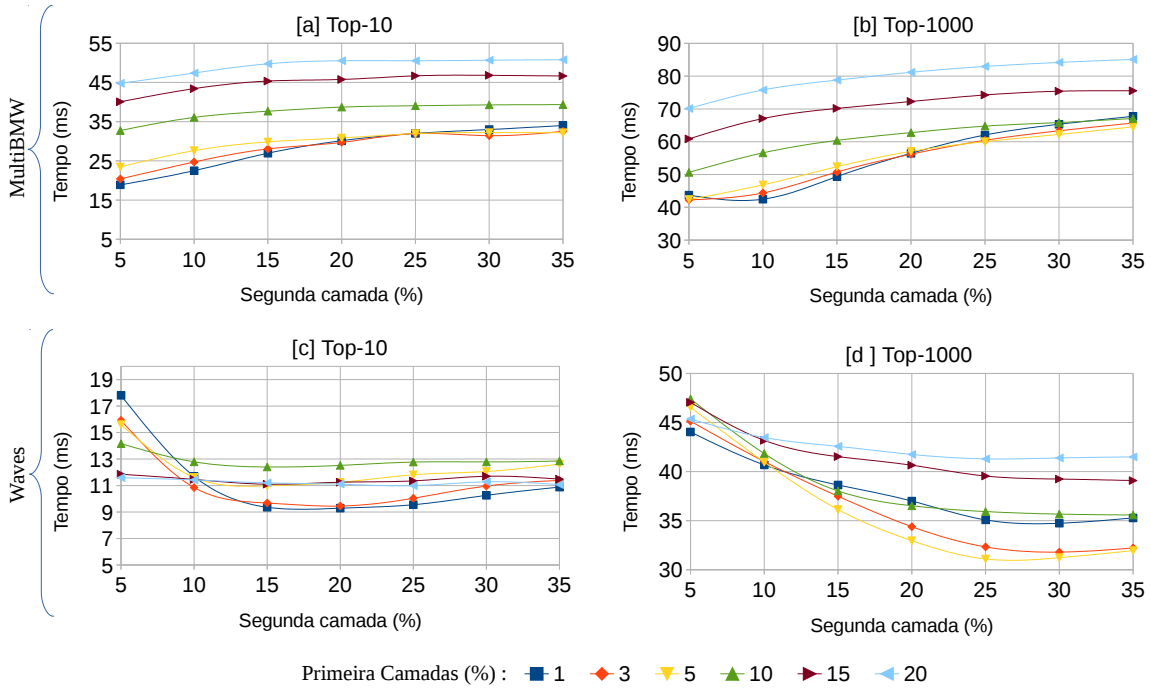

Figura 5.2: Três camadas: Variação do tempo de processamento dos métodos para diferentes divisões do índice em camadas. Processando top-10 e top-1000 respostas.

dos métodos Waves e MBMW para cada variação do índice em três camadas pode ser visto nos gráficos da Figura 5.2

Com a divisão do índice em três camadas os melhores tempos encontrados para os métodos MBMW e Waves ao computar o ranking top-1000 são 42.2 *ms* e 31.1 *ms* respectivamente. As melhores divisões em camadas ainda na mesma ordem para top-1000 são: primeira camada $\Delta = 3\%$, segunda camada $\Delta = 5\%$ e primeira camada $\Delta = 5\%$, segunda camada $\Delta = 25\%$.

Ao processar top-10 respostas, os melhores tempos são 18.8 *ms* para o método MBMW e 9.2 *ms* para o método Waves. Os índices referentes aos melhores tempos são : primeira camada $\Delta = 1\%$, segunda camada $\Delta = 5\%$ e primeira camada $\Delta = 1\%$, segunda camada $\Delta = 20\%$.

Os melhores tempos e configurações para cada método, no que diz respeito à divisão do índice em camadas, estão apresentados na Tabela 5.1. Foram realizados ainda alguns experimentos para avaliar o tempo do método Waves com índice em quatro camadas, porém não foram obtidos resultados melhores. O método Waves obteve seu melhor desempenho em tempo de processamento com índice em 3 camadas, sendo também o mais rápido método de processamento de consultas nos experimentos realizados até aqui. As

	Tempo(ms)	#Camadas	Divisão
	top-10		
WAND	46,23	1	100%
BMW	24,03	1	100%
MBMW	17,02	2	4%, 96%
BMW-CSP	10,87	2	24%, 76%
Waves	9,29	3	1%, 20% 79%
	top-1000		
WAND	75,30	1	100%
BMW	57,10	1	100%
MBMW	40,02	2	8%, 92%
BMW-CSP	39,89	2	42%, 58%
Waves	31,10	3	5%, 25% 70%

Tabela 5.1: Tempo médio e a melhor divisão em camadas por método para processar mil consultas na coleção Gov2.

configurações de divisão do índice em camadas apresentadas na Tabela 5.1 serão usadas nos próximos experimentos apresentados na seção seguinte com a coleção de consultas de teste.

5.2.1 Coleções menores [Gov2-1 e Gov2-10]

Nesta seção avaliamos o impacto do tamanho da coleção para cada método. O tamanho da coleção pode afetar a escolha do melhor método ou da melhor divisão do índice em camadas. Por exemplo, listas invertidas menores têm menos blocos, e assim as heurísticas para descartar blocos podem ser afetadas.

	Gov2	Gov2-10	Gov2-1
	top-10		
MBMW	4%	2%	2%
BMW-CSP	24%	20%	20%
Waves	20%	14%	10%
	top-1000		
MBMW	8%	4%	8%
BMW-CSP	42%	44%	30%
Waves	14%	8%	12%

Tabela 5.2: Melhor tamanho da primeira camada, entre as variações do índice em duas camadas, para cada método em cada coleção.

Na seção anterior foram apresentados experimentos para encontrar a melhor divisão em camadas para cada método na coleção Gov2 completa. Nesta seção apresentamos experimentos semelhantes, porém para encontrar a melhor divisão em camadas considerando duas variações da coleção Gov2. A primeira variação contém um milhão de

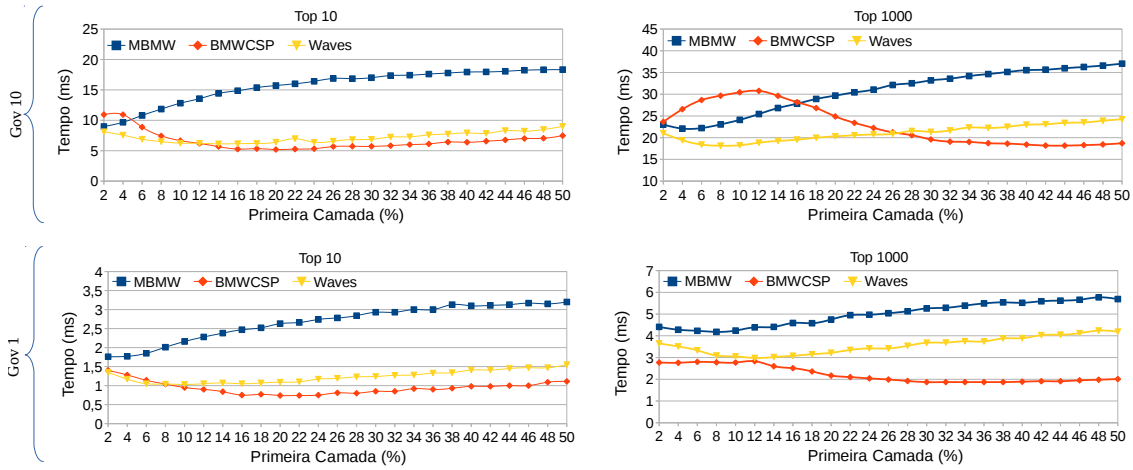

Figura 5.3: Variação do tempo de processamento dos métodos para diferentes divisões do índice em duas camadas, nas coleções Gov2-10, com dez milhões de documentos, e Gov2-1, com um milhão de documentos. Processando top-10 e top-1000 respostas.

documentos e chamamos de Gov2-1. A segunda contém dez milhões de documentos e chamamos de Gov2-10. Os índices gerados para cada coleção também foram divididos em 25 versões com duas camadas e 42 versões com três camadas, como apresentado na seção anterior para a coleção Gov2 completa. O resultado dos experimentos com a divisão do índice em duas camadas, nas duas coleções de documentos, são apresentados nos gráficos da Figura 5.3. Os resultados dos experimentos com a divisão do índice em três camadas são apresentados nos gráficos das Figuras 5.4 e 5.5.

Uma característica que pode ser observada nos gráficos da Figura 5.3, em relação à escolha do melhor método considerando o tamanho da coleção, é que as curvas de cada método se afastam à medida que o tamanho da coleção de documentos diminui. No gráfico top-1000 da coleção Gov2-1, as três curvas já não se tocam. Essa característica indica que, independente da proporção em que o índice é dividido, o método BMW-CSP é a melhor opção em tempo de processamento quando se trata de coleções menores.

O ponto ótimo de divisão do índice é diferente em diferentes tamanhos da coleção de documentos. A Tabela 5.2 mostra a melhor divisão do índice em duas camadas para cada método em cada coleção de documentos. Apesar do ponto ótimo variar entre as coleções de documentos, o tempo de processamento para as diferentes versões do índice nas coleções menores varia pouco. Por exemplo, se assumirmos o ponto ótimo de divisão do índice na coleção Gov2 para a coleção Gov2-1 o tempo dos métodos nesta coleção vão

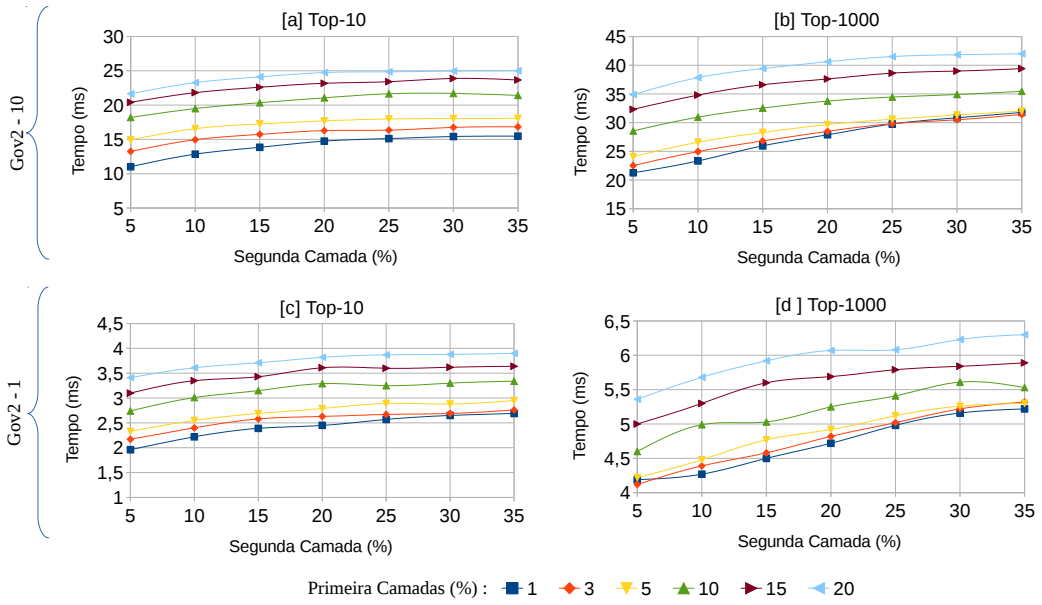

Figura 5.4: Variação do tempo de processamento do método MBMW para diferentes divisões do índice em três camadas, nas coleções Gov2-10 com dez milhões de documentos e Gov2-1 com um milhão de documentos. Processando top-10 e top-1000 respostas.

variar de 4.1 *ms* para 4.1 *ms*, 1.8 *ms* para 1.9 *ms* e 2.9 *ms* para 3.0 *ms* respectivamente para os métodos MBMW, BMWCSP e Waves.

Diferente dos experimentos com as coleções Gov2 e Gov2-10, mesmo dividindo o índice da coleção Gov2-1 em três camadas, o melhor tempo de processamento do método Waves não supera o tempo alcançado com o método BMW-CSP nessa coleção. Na medida em que a coleção aumenta, o método Waves passa a ser a melhor opção em tempo de processamento e em quantidade de memória utilizada.

	Gov2	Gov2-10	Gov2-1
	top-10		
WAND	46,23	18,95	2,18
BMW	24,03	11,37	2,85
MBMW	17,02	8,98	1,76
BMW-CSP	10,87	5,20	**0,74**
Waves	**9,29**	**4,94**	0,83
	top-1000		
WAND	75,30	33,45	5,21
BMW	57,10	27,58	5,17
MBMW	40,02	22,09	4,18
BMW-CSP	38,89	18,17	**1,87**
Waves	**31,10**	**16,37**	2,93

Tabela 5.3: Tempo médio em milissegundos por método para processar mil consultas nas coleção Gov2, Gov2-10 e Gov2-1. Os menores valores são apresentados negrito

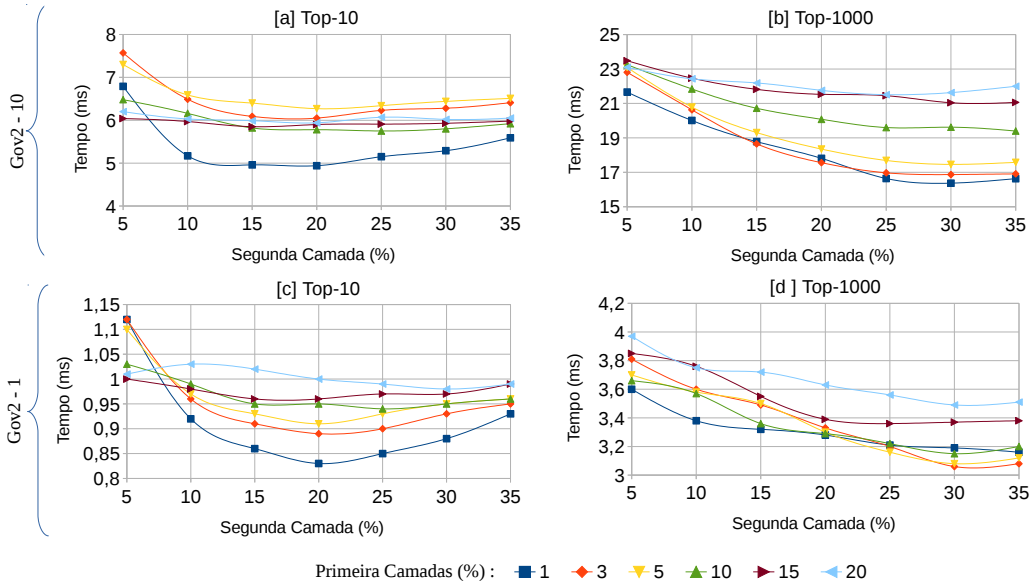

Figura 5.5: Variação do tempo de processamento do método Waves para diferentes divisões do índice em três camadas, nas coleções Gov2-10 com dez milhões de documentos e Gov2-1 com um milhão de documentos. Processando top-10 e top-1000 respostas.

A Tabela 5.3 apresenta os melhores tempos de todos os métodos avaliados para cada variação da coleção Gov2 ao retornar top-10 e top-1000 documentos na resposta. Os melhores tempos para cada coleção são apresentados em negrito, indicando o método que possui melhor desempenho.

5.3 Avaliação dos parâmetros de divisão do índice

O principal objetivo dos experimentos nesta seção é verificar se o melhor parâmetro para divisão do índice selecionado para o conjunto de consultas de calibração produz resultados competitivos ao processar o conjunto de consultas de teste. A seleção do tamanho de cada camada e do número de camadas é um passo importante quando se utilizam índices de múltiplas camadas, uma vez que esse parâmetro pode afetar o desempenho final dos métodos.

A Tabela 5.4 apresenta o desempenho de cada método ao processar as dez mil consultas. Os dados apresentados na tabela são: a média do tempo de processamento, a média do número de blocos acessados e a média do número de pivôs selecionados durante o processamento. Há uma variação natural do número médio de operações e do tempo médio de processamento entre cada coleção de consultas. Essa variação implicou em um

desempenho ainda melhor do método Waves e do método BMW-CSP sobre os *baselines*.

Por exemplo, em comparação com MBMW, Waves conseguiu uma redução de 54% no tempo de processamento dos top-10 resultados (de 20,52 ms a 9,45 ms) ao processar o conjunto de consultas de teste. Essa redução superou a alcançada durante o processamento do conjunto de consultas de calibração, que era de 48%. Ao olhar para os tempos alcançados ao processar as top-1000 respostas, o ganho no conjunto de consultas de teste foi de 30,1% (de 47,23ms para 33,01 ms), a mesma alcançada no conjunto de consulta de calibração.

Para conferir se os parâmetros selecionados para o conjunto de consultas de calibração também são os melhores para o conjunto de consultas de teste, nós variamos os parâmetros do conjunto de consultas de teste e percebemos que todos os parâmetros resultantes do conjunto de calibração são ótimos ou muito perto dos parâmetros ideais. Isso indica que parâmetros podem ser ajustados usando consultas de exemplo.

	#Bloco	#Pivô	Tempo (ms)
	top-10		
Wand	30.245	1.004.861	$44,489 \pm 1,21$
BMW	20.305	464.492	$24,884 \pm 0,84$
MBMW	16.107	331.806	$20,621 \pm 0,77$
BMW-CSP	5.582	201.559	$10,730 \pm 0,61$
Waves	8.099	144.130	$9,453 \pm 0,56$
	top-1000		
Wand	35.705	1.856.409	$76,904 \pm 1,72$
BMW	32.710	1.189.637	$59,336 \pm 1,47$
MBMW	28.895	835.791	$47,226 \pm 1,34$
BMW-CSP	17.580	688.172	$39,731 \pm 1,26$
Waves	26.168	402.196	$33,006 \pm 1,20$

Tabela 5.4: Número médio de blocos acessados, número médio de pivôs avaliados e desempenho médio de tempo com intervalo de confiança de 95%. Usando Wand, BMW, MBMW,BMW-CSP e Waves ao processar o maior conjunto de consultas, usando os melhores parâmetros encontrados para o menor conjunto de consultas.

5.3.1 Número de termos na consulta

Nesta seção apresentamos os resultados da avaliação dos métodos ao considerar o número de termos nas consultas processadas. Para isso, separamos o conjunto de consultas de teste em seis grupos. As consultas com até cinco termos foram separadas em cinco grupos diferentes e as consultas com mais de cinco termos compõem o sexto grupo.

O desempenho de cada método ao processar cada um dos seis grupos de consultas é apresentado nos gráficos da Figura 5.6. Os tempos alcançados com o método Waves são menores em todos os tamanhos de consulta, computando as top-10 e as top-1000 respostas. A maior diferença ao comparar o método MBMW com o método Waves ocorre no conjunto das consultas com um termo. Essa diferença maior pode ser explicada pela seleção do tamanho das camadas em cada método. A primeira camada no método Waves é menor do que a primeira camada do método MBMW. Assim, o tempo esperado para o processamento de consultas com um único termo no método Waves tornou-se ainda menor em relação ao *baseline*. Ressalta-se que o conjunto menor de consultas representa uma pequena parte do total de consultas experimentadas, menos de 2%, o que explica por que essa diferença nesse grupo afeta pouco o tempo médio de processamento dos métodos.

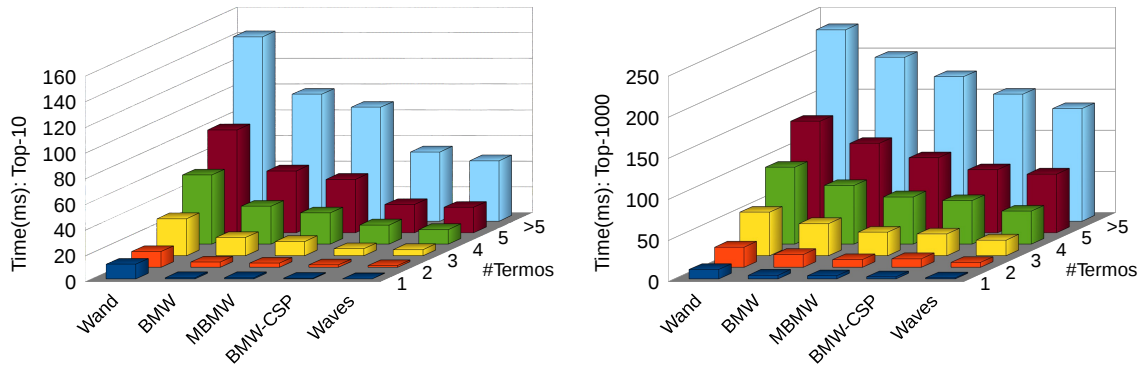

Figura 5.6: Tempo médio de processamento dos métodos ao processar consultas com tamanhos distintos extraídas da coleção de consultas de teste, computando top 10 e top 1000 resposta.

5.3.2 Número de waves

O objetivo dos experimentos apresentados nesta seção é comparar o tempo dos métodos ao processar as consultas que exigem mais ou que exigem menos processamento dos métodos. Para isso, nós agrupamos as consultas de acordo com o número de waves exigidas pelo método Waves com sua melhor configuração e comparamos o desempenho com os tempos obtidos com os *baselines*.

Os resultados dos experimentos propostos nesta seção estão representados na Tabela 5.5. Poucas consultas necessitaram de três waves para serem processadas. Para obter

as top-10 respostas, apenas 5 consultas usaram a terceira wave e para obter as top-1000 respostas, 9 consultas. O método Waves foi o mais lento para processar essas consultas específicas, mas o custo extra para processá-las não é muito significativo, pois essas consultas também são as mais caras para os *baselines*. Por exemplo, o tempo obtido com o método MBMW ao computar as cinco consultas que exigem três waves no cenário top-10 é 684 *ms*, enquanto no Waves o tempo é 690 *ms*.

Uma observação sobre as consultas que vão para a terceira wave é que são consultas longas, chegando até a 18 termos. Além disso, como podemos ver nos experimentos, elas são raras. Menos de 0,1 % das consultas requerem a terceira wave. Notamos que, mesmo para essas consultas, que representam a pior situação para o método Waves, o algoritmo ainda alcança desempenho competitivo quando comparado com os *baselines*.

		Tempo (ms)				
# waves	#queries	Wand	BMW	MBMW	BMW-CSP	Waves
top-10						
1	2794	10,26	2,75	2,83	0,61	**0,52**
2	7201	58,56	33,18	27,27	14,53	**12,44**
3	5	**618,00**	638,00	684,00	640,00	690,00
top-1000						
1	1662	18,41	11,97	7,94	9,33	**4,76**
2	8329	88,95	70,42	55,80	42,95	**37,56**
3	9	867,77	803,66	850,00	**700,22**	1036,66

Tabela 5.5: Tempo dos métodos para o processar consultas agrupadas de acordo com o número de waves exigido pelo método Waves.

5.4 Experimentos com stopwords

A configuração dos experimentos para esta seção considera a indexação da coleção de documentos sem remover as *stopwords*. Esse é o cenário mais comum para um processador de consultas em um ambiente real. Para avaliar o desempenho dos métodos nesse novo cenário utilizamos os mesmos experimentos adotados até a seção anterior, incluindo uma nova verificação de melhor ponto de divisão do índice para cada método.

Nestsa seção, também apresentamos uma versão otimizada dos *baselines* BMW e MBMW, que chamamos de BMWT e MBMWT. Na nova versão os *baselines* adotam o mesmo limiar inicial proposto nesta tese para os métodos BMW-CSP e Waves.

Na Tabela 5.6, são apresentados os resultados alcançados com os métodos para o cená-

rio proposto nesta seção e o melhor percentual de separação do índice para cada método. Os resultados apresentados na Tabela 5.6 são: tempo médio de processamento, número médio de pivôs e número médio de blocos acessados por cada método para processar o maior conjunto de consultas sem remover as *stopwords* e utilizando o índice da coleção Gov2.

Os experimentos para encontrar o melhor ponto de divisão do índice nesse novo cenário levaram a parâmetros próximos aos de divisão do índice que já haviam sido definidos no cenário sem *stopwords*. Ainda assim, adotamos os novos parâmetros, que são reportados na Tabela 5.6, para todos os próximos experimentos.

	#Blocos	#Pivos	Tempo (ms)	Divisão
	top-10			
BMW	39.447	633.998	$47,56 \pm 2,09$	100%
BMWT	38.455	613.341	$46,21 \pm 2,09$	100%
MBMW	26.713	438.833	$36,29 \pm 1,92$	5%, 95%
MBMWT	25.958	422.986	$35,27 \pm 1,91$	5%, 95%
BMW-CSP	11.028	313.153	$19,64 \pm 1,35$	30% 70%
Waves	14.577	176.709	**16,08** $\pm 1,03$	1% 20% 79%
	top-1000			
BMW	81.590	1.712.494	$115,71 \pm 3,71$	100%
BMWT	79.329	1.580.775	$108,28 \pm 3,69$	100%
MBMW	60.733	1.128.392	$89,23 \pm 3,16$	10%, 90%
MBMWT	58.053	1.021.467	$82,66 \pm 3,14$	10%, 90%
BMW-CSP	37.241	910.131	$66,35 \pm 2,92$	40%, 60%
Waves	49.363	579.925	**57,40** $\pm 2,55$	5% 30% 65%

Tabela 5.6: Número médio de blocos acessados, número médio de pivôs avaliados e desempenho médio de tempo(ms) com intervalo de confiança de 95% usando BMW, BMWT, MBMW, MBMWT, BMW-CSP e Waves ao processar o maior conjunto de consulta no índice com *stopwords*.

O impacto de manter as *stopwords* no processamento de consultas é percebido ao comparar os resultados da Tabela 5.6 com os resultados da Tabela 5.4. Manter as *stopwords* nas consultas levou ao aumento médio no tempo de processamento de 77,5% para top-10 e 75,6% para top-1000. Aumentando por exemplo de $39,7$ *ms* para $66,3$ *ms* e de $33,0$ *ms* para $57,4$ *ms* respectivamente os tempos dos métodos BMW-CSP e Waves ao computar os top-1000 resultados.

A redução do tempo de processamento ao comparar o método Waves com o melhor baseline (MBMWT) é de aproximadamente 54,5% para o top-10 e de 30,5% para o top-1000. Essa redução é aproximadamente a mesma que pode ser observada na Tabela 5.4 ao comparar os mesmos métodos sem *stopwords*.

Os gráficos da Figura 5.7 possibilitam comparar tempo requerido para as duas primeiras fases do método BMW-CSP com as duas fases exigidas no método BMW-CS. Estes experimentos são úteis para quantificar as melhorias nas duas primeiras fases obtidas pelas mudanças que propusemos. A Figura 5.7(a) e a Figura 5.7(c) mostram que o BMW-CSP obteve um desempenho de tempo melhor do que o BMW-CS para computar consultas tanto para top-10 quanto para top-1000. Também vemos uma variação no número de candidatos selecionados na primeira fase ao comparar os métodos. Ao examinar o número médio de candidatos necessários para o processamento de consultas, vemos que esse número diminui cerca de 20% para top-10 e cerca de 30% ao computar top-1000 resultados.

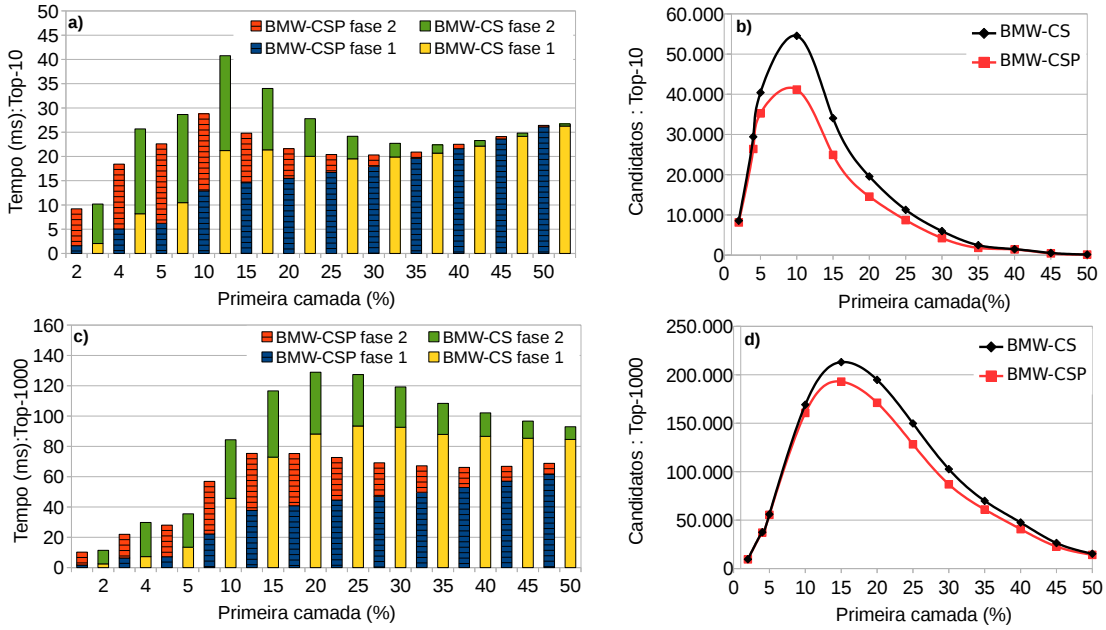

Figura 5.7: Tempo médio e número médio de candidatos obtidos com o método BMW-CS e com as duas primeiras fases do método BMW-CSP ao variar o tamanho da primeira camada computando os top-10 (a e b) e os top-1000 resultados (c e d).

A Figura 5.7 também mostra um comportamento interessante da segunda fase dos dois métodos, considerando a evolução do número de candidatos à medida que aumenta o tamanho da primeira camada. Quando a primeira camada é muito pequena, ex. com 2% do índice, o número de candidatos é pequeno. Neste nível, muitos documentos na segunda camada têm chance de estar entre os top-k, já que muitos documentos com escore alto podem não ocorrer na primeira camada. À medida que aumentamos o tamanho da primeira camada de 2% para 10% ao calcular top-10 e para 15% ao calcular top-1000,

o número de candidatos aumenta, uma vez que temos mais documentos ocorrendo na primeira camada. Quando continuamos aumentando o tamanho da primeira camada, as estimativas de escore ficam mais precisas, permitindo que mais candidatos sejam descartados. Como consequência, o número de candidatos diminui para tamanhos maiores da primeira camada. Em um limite, se fizermos a primeira camada com todo o índice, a seleção de candidatos seria capaz de escolher apenas o conjunto exato de documentos para compor as respostas, minimizando o tamanho da lista de candidatos. Assim, o conjunto de candidatos é menor quando a primeira camada é muito pequena, aumenta até tamanhos intermediários e diminui novamente à medida que a primeira camada cresce.

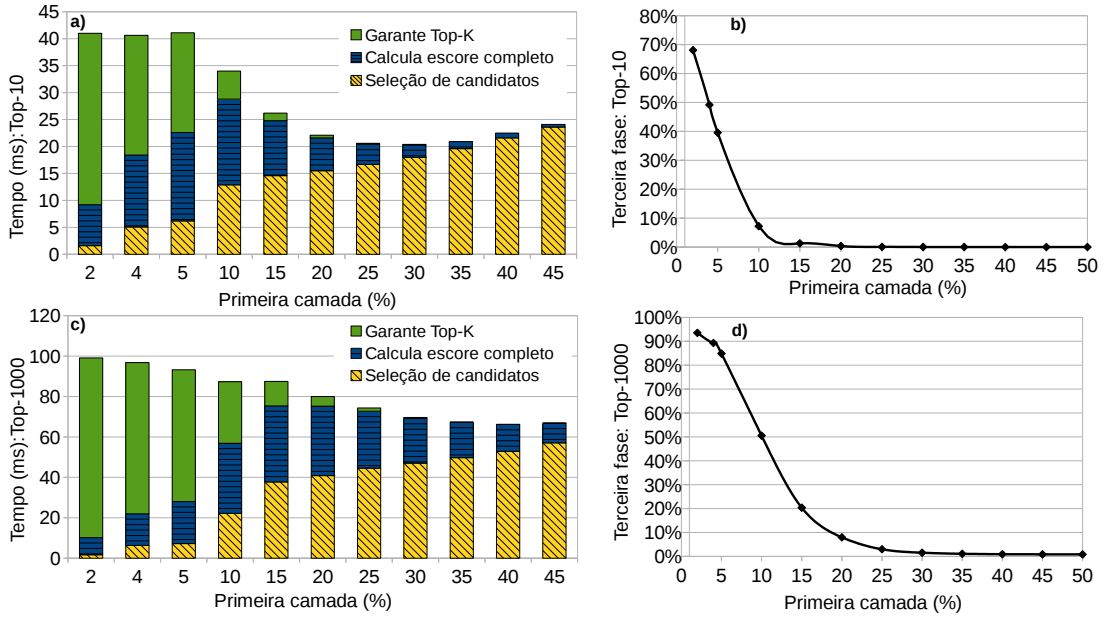

Figura 5.8: Tempo médio de cada fase do método BMW-CSP ao variar o tamanho da primeira camada do índice e número de médio de execução da terceira fase ao computar top-10 (a e b) e top-1000 resultados (c e d).

Os gráficos da Figura 5.8 apresentam o tempo requerido por cada uma das três fases do método BMW-CSP para diferentes tamanhos da primeira camada ao computar top-10 (a) e top-1000 (c) respostas.

Observando a evolução do número de candidatos apresentados na Figura 5.7, vemos que o tempo necessário para a primeira fase do processamento é afetado por dois principais fatores, o número de candidatos selecionado e o tamanho da primeira camada. A segunda fase é afetada pelo número de candidatos, que cresce até um certo ponto de divisão do índice e em seguida diminui. A terceira fase tem um comportamento distinto das

duas primeiras, à medida que o tamanho da primeira camada aumenta, diminui a chance de um documento da segunda camada compor a resposta e o número de consultas que vão para a terceira fase cai rapidamente, como mostrado nas Figuras 5.8 (b e d).

Os gráficos apresentados nas Figuras 5.9, 5.10 e 5.11 apresentam uma visão geral do desempenho dos métodos MBMWT, BMW-CSP e Waves para cada consulta do maior conjunto de consultas. Os gráficos da Figura 5.9 possibilitam a comparação entre o desempenho do método BMW-CSP e do melhor baseline, MBMWT. Os gráficos da Figura 5.10 possibilitam a mesma comparação entre o método Waves e o baseline MBMWT. Já os gráficos da Figura 5.11 possibilitam uma comparação entre o desempenho dos dois métodos propostos, os métodos Waves e BMW-CSP.

Nos gráficos da Figura 5.9, as consultas são classificadas segundo o tempo para a execução do método BMW-CSP, que permite uma distinção clara entre o seu desempenho e o desempenho do método MBMWT. Pontos acima da linha formada pelos resultados do BMW-CSP indicam consultas onde o BMW-CSP foi melhor do que o MBMWT. Tempos obtidos ao computar os top-10 resultados são apresentados na Figura 5.9(a) e os obtidos ao computar os top-1000 resultados são apresentados na Figura 5.9(b).

Para permitir uma melhor visão dos pontos nos gráficos, apresentamos gráficos separados para consultas que levam menos de 1 segundo (a) e (b), e para consultas que levam mais de 1 segundo (c) e (d). Consultas que levam mais de 1 segundo representam um pequeno conjunto em ambos os cenários, sendo exatamente 15 consultas no cenário top-10 e 49 consultas no cenário top-1000. Os gráficos são úteis para mostrar que o método BMW-CSP foi mais rápido ao processar a maioria das consultas para ambos os resultados, top-10 e top-1000.

Ao analisar o conjunto consultas dos gráficos (c) e (d), percebemos que eles são em sua maioria resultados de consultas grandes, com uma variação de 9 a 25 termos nessas consultas. Comparando o desempenho dos métodos para essas consultas, o BMW-CSP foi ligeiramente melhor do que MBMWT ao calcular os top-10 resultados. Enquanto para top-1000, o BMW-CSP foi pior em 26 consultas de um total de 49. A manutenção do conjunto de documentos candidatos em consultas longas retarda o processamento do BMW-CSP, especialmente ao calcular os top-1000 resultados. Uma conclusão clara é que a BMW-CSP não é uma boa opção em aplicações onde se espera que o número de termos seja muito alto. Notamos que em tais situações, métodos de processamento de

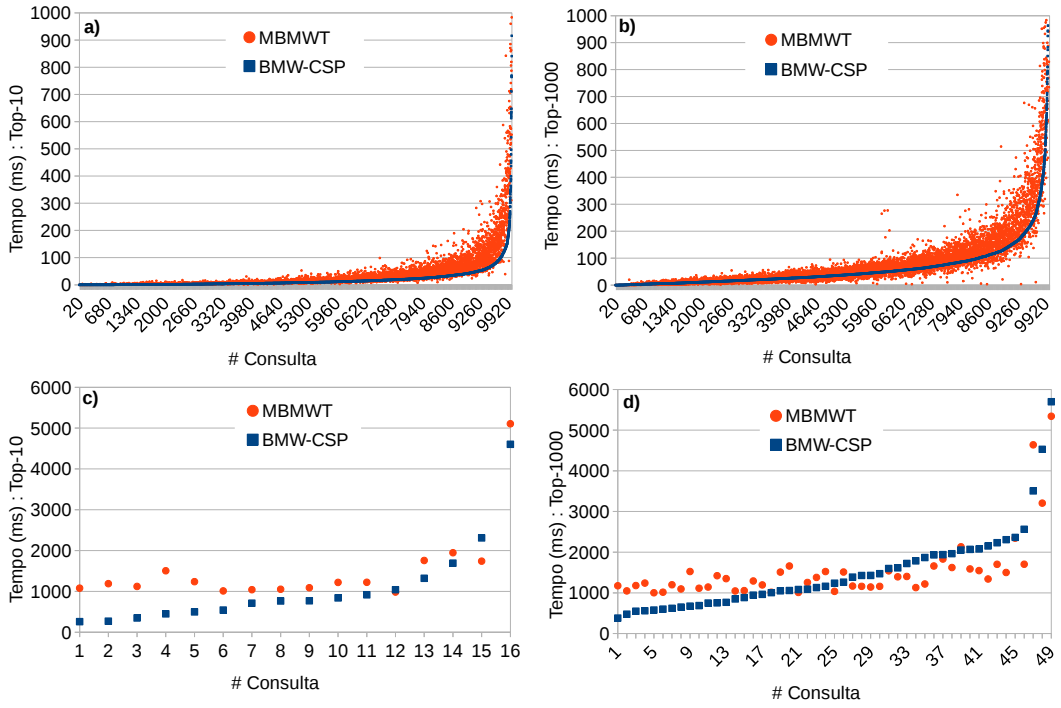

Figura 5.9: Tempo de processamento dos métodos BMW-CSP e MBMW para cada consulta ao computar o ranking top-10(a) e top-1000(b). Para melhor visualização do gráfico, as consultas que levam mais de 1 segundo foram separadas, para top-10 (c) e para top-1000 (d).

consultas documento a documento não são recomendados como a melhor opção para o processamento de consultas e que os métodos termo a termo são geralmente as melhores soluções.

Nos gráficos da Figura 5.10, as consultas são classificadas segundo o tempo para a execução do método Waves, permitindo uma distinção entre o seu desempenho e o desempenho do método MBMWT. Resultados do processamento top-10 com menos de $0,5$ *ms* são apresentados na Figura 5.10(a) e com mais de $0,5$ *ms* na Figura 5.10(c). Os resultados obtidos ao computar as top-1000 respostas são apresentados nas Figuras 5.10(b) e 5.10(d). Os gráficos são úteis para mostrar que o Waves foi mais rápido ao processar a maioria das consultas para ambos os cenários top-10 e top-1000.

Ao comparar os gráficos apresentados na Figura 5.10 com os gráficos apresentados na Figura 5.9 pode-se perceber que há um maior volume de pontos acima da linha de referência nos gráficos da Figura 5.10, reforçando os resultados já verificados que indicam que o método Waves tem o melhor desempenho.

Os gráficos da Figura 5.11 apresentam uma comparação semelhante, mas agora com-

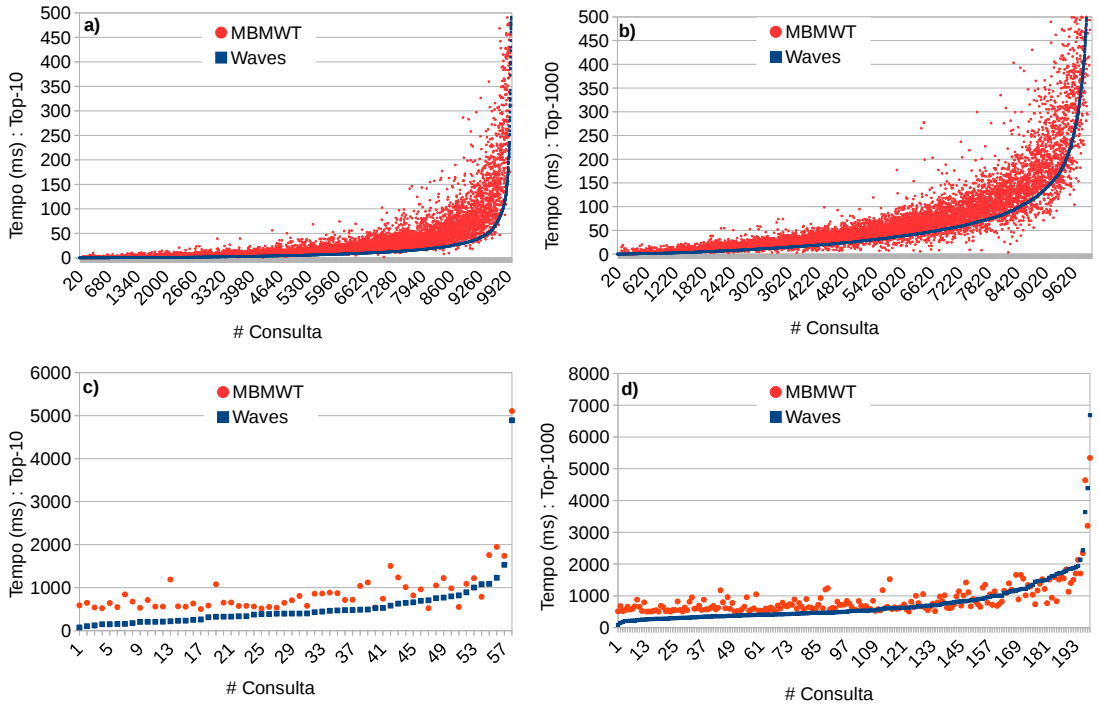

Figura 5.10: Tempo de processamento dos métodos Waves e MBMWT para cada consulta ao computar o ranking top-10(a) e top-1000(b). Para melhor visualização do gráfico, as consultas que levam mais de 0,5 segundo foram separadas, para top-10 (c) e para top-1000 (d).

parando o método Waves com o método BMW-CSP. Mais uma vez as consultas são classificadas segundo o tempo para a execução do método Waves, e para melhor visualização do gráfico as consulta que levaram mais de $0,5$ *s* para serem processadas foram plotadas em gráficos separados (c) e (d), para top-10 e top-1000 respectivamente.

Consultas que levam mais de $0,5$ segundos representam um pequeno conjunto em ambos os cenários, o que constitui exatamente 16 consultas (0,16 %) no cenário de top-10, e 99 consultas (0,99 %) no cenário de Top-1000. As figuras são úteis para mostrar que Waves foi claramente mais rápido do que o BMW-CSP ao processar a maioria das consultas de top-10. Ao olhar para o cenário de top-1000, o método Waves ainda é mais rápido para a maioria das consultas, apesar de a diferença entre os dois métodos ser menor.

A Tabela 5.7 descreve o número de consultas em que o método Waves é mais lento, mais rápido ou tem quase o mesmo tempo do que o *baseline* MBMWT e o método BMW-CSP. Consideramos um empate quando a diferença é inferior a $0,01$ *ms*. Para processar top-10, o modo Waves foi mais rápido do que o BMW-CSP em cerca de 91% das consultas e mais rápido do que o MBMWT em cerca de 98% das consultas. Para top-1000, Waves

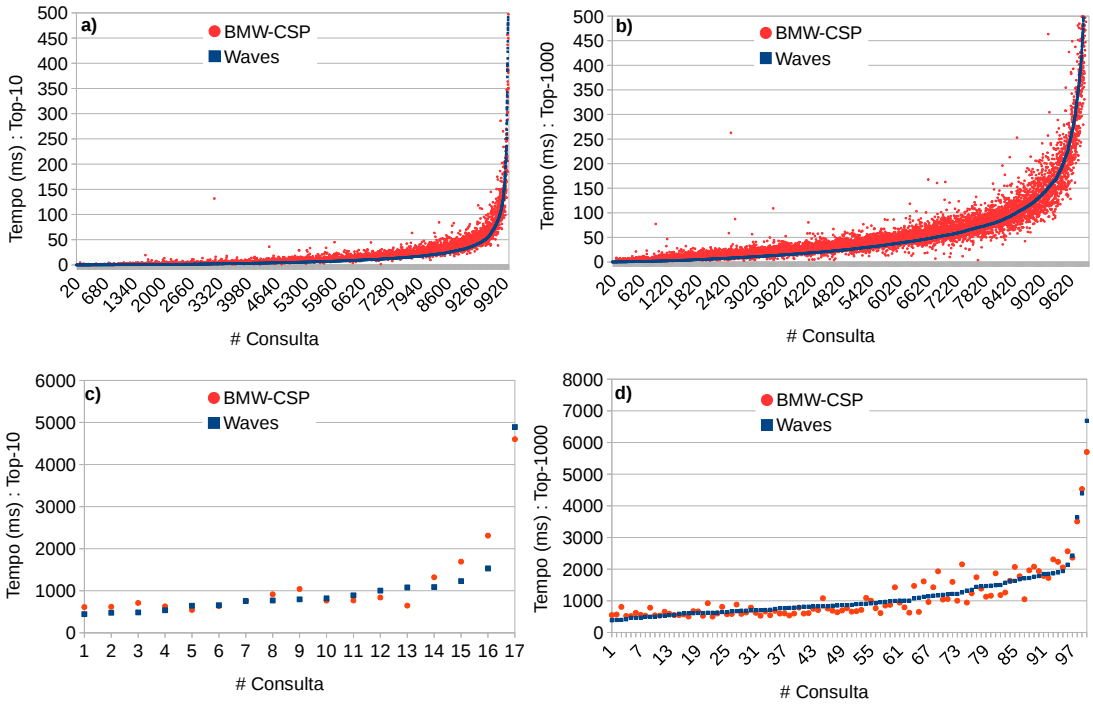

Figura 5.11: Tempo de processamento dos métodos Waves e BMW-CSP para cada consulta ao computar o ranking top-10(a) e top-1000(b). Para melhor visualização do gráfico, as consultas que levam mais de 0,5 segundo foram separadas, para top-10 (c) e para top-1000 (d).

foi mais rápido do que o BMW-CSP em 85,2% das consultas e é mais rápido do que o MBMWT em cerca de 91%.

#Baseline	Top 10	Top 1000
	Waves ganha	
BMW-CSP	9.088	8.521
MBMWT	9.747	9.101
	Waves perde	
BMW-CSP	749	1.403
MBMWT	159	827
	Empate	
BMW-CSP	163	76
MBMWT	94	72

Tabela 5.7: Número de consultas que Waves é mais rápido ou mais lento do que os *baselines* BMW e BMW-CSP. Empate: Quando a diferença é inferior a 0.01ms.

Na Tabela 5.8 são apresentados os desempenhos dos métodos para diferentes números de termos nas consultas. Podemos ver que a vantagem do método Waves reduz com o aumento do tamanho das consultas. À medida que se aumenta o número de termos em uma consulta, o número esperado de waves para processar a consulta também aumenta,

pois a chance de encontrar um documento em um dos níveis de um termo da consulta, e não encontrar o mesmo documento no mesmo nível para outros termos da consulta, também aumenta.

Algorithm	Tempo (ms)					
	1	2	3	4	5	+5
#Queries	195	1.484	2.498	2.388	1.599	1.836
	top-10					
BMW	$0,56 \pm 0,17$	$3,72 \pm 0,29$	$14,70 \pm 0,70$	$30,03 \pm 1,20$	$53,12 \pm 2,42$	$151,70 \pm 9,60$
BMWT	$0,41 \pm 0,16$	$3,35 \pm 0,29$	$13,77 \pm 0,71$	$28,61 \pm 1,22$	$51,42 \pm 2,43$	$149,15 \pm 9,62$
MBMW	$0,45 \pm 0,10$	$2,70 \pm 0,21$	$11,12 \pm 0,63$	$22,19 \pm 1,02$	$38,97 \pm 2,13$	$118,02 \pm 9,23$
MBMWT	$0,37 \pm 0,09$	$2,44 \pm 0,21$	$10,58 \pm 0,64$	$21,30 \pm 1,03$	$37,68 \pm 2,14$	$115,81 \pm 9,22$
BMW-CSP	$0,19 \pm 0,06$	$1,80 \pm 0,14$	$6,53 \pm 0,30$	$12,56 \pm 0,50$	$20,58 \pm 0,91$	$62,65 \pm 6,93$
Waves	**0,08** $\pm 0,01$	**1,02** $\pm 0,09$	**4,21** $\pm 0,21$	**8,85** $\pm 0,39$	**15,31** $\pm 0,65$	**56,00** $\pm 5,80$
	top-1000					
BMW	$2,94 \pm 0,63$	$13,040 \pm 0,66$	$42,77 \pm 1,48$	$80,17 \pm 2,36$	$133,92 \pm 4,49$	$342,59 \pm 15,29$
BMWT	$1,64 \pm 0,31$	$9,51 \pm 0,62$	$37,00 \pm 1,46$	$72,59 \pm 2,34$	$124,87 \pm 4,49$	$330,52 \pm 15,34$
MBMW	$2,32 \pm 0,55$	$9,99 \pm 0,54$	$32,96 \pm 1,25$	$62,10 \pm 1,91$	$102,26 \pm 3,64$	$264,51 \pm 13,80$
MBMWT	$1,52 \pm 0,41$	$7,50 \pm 0,49$	$28,24 \pm 1,22$	$55,36 \pm 1,89$	$94,32 \pm 3,66$	$252,94 \pm 13,85$
BMW-CSP	$1,38 \pm 0,27$	$9,43 \pm 0,54$	$25,77 \pm 0,79$	$44,83 \pm 1,26$	$70,10 \pm 2,21$	$200,59 \pm 14,01$
Waves	**0,42** $\pm 0,10$	**4,73** $\pm 0,31$	**18,56** $\pm 0,53$	**36,23** $\pm 0,98$	**61,47** $\pm 1,93$	**182,89** $\pm 13,98$

Tabela 5.8: Tempo médio com intervalo de confiança de 95% para processar consultas com diferentes números de termos ao computar o ranking top-10 e top-1000.

Esse fenômeno é ilustrado na Tabela 5.9, onde podemos ver a evolução do número de waves necessárias quando o tamanho da consulta varia. Podemos ver que consultas com mais termos são suscetíveis a exigir mais waves para serem processadas, o que afeta o desempenho do Waves.

De qualquer forma, também vemos melhorias no desempenho para todos os tamanhos de consulta ao comparar o Waves com o MBMWT. A maior diferença nos tempos ao comparar o MBMWT e o BMW-CSP com o método Waves ocorre no conjunto de consultas com 1 termo e pode ser explicada pela seleção dos melhores tamanhos das camadas em cada método. A primeira camada no método Waves foi menor do que a primeira camada no BMW-CSP e no MBMWT na escolha da melhor divisão. Assim, o tempo esperado para o processamento de consultas com Waves tornou-se menor em consultas com apenas 1 termo. Ressalta-se que consultas com 1 termo representam uma pequena porção das consultas totais neste conjunto de dados, o que explica por que tal diferença enorme em tempos não afetou os tempos médios de processamento de consultas dos métodos. Enquanto o método Waves é melhor do que os *baselines* para todos os tamanhos de consulta

experimentados, os resultados apresentados na Tabela 5.8 indicam que os ganhos obtidos com o método Waves são menores para consultas maiores.

	# Consultas					
waves	1	2	3	4	5	+5
	top-10					
1	195(100%)	1.173(79%)	890 (36%)	399 (17%)	128 (8%)	42 (2%)
2	0 (0%)	311(21%)	1.608 (64%)	1.989 (83%)	1.471 (91%)	1.766 (96%)
3	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	28 (2%)
	top-1000					
1	195 (100%)	875 (58%)	377 (15%)	133 (5%)	27 (2%)	3 (0%)
2	0 (0%)	609 (42%)	2.121 (85%)	2.255 (95%)	1.572 (98%)	1.812(98%)
3	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	21 (2%)

Tabela 5.9: Distribuição das consultas de acordo com o número de termos e o número de waves necessárias para o processamento, ao computar os resultados top-10 e top-1000.

Uma questão que pode surgir é quanto ao tempo alcançado pelo método Waves para processamento que exige mais waves. Para responder a essa pergunta, agrupamos as consultas de acordo com o número de waves necessárias no método Waves com sua melhor configuração e comparamos o desempenho dos diferentes métodos em cada grupo de consultas. Os resultados estão representados na Tabela 5.10. Podemos ver que poucas consultas necessitam de três waves para serem processadas, sendo 28 consultas ao computar top-10, e 21 consultas ao computar top-1000. Essas consultas representam um caso difícil para o método Waves, mas o custo extra para processá-las não é proibitivo, quando comparado aos tempos obtidos pelos demais métodos. Por exemplo, BMW-CSP foi o melhor método ao computar as 28 consultas que exigem três waves no cenário top-10. Seu tempo foi no entanto bastante próximo ao obtido pelo método Waves, sendo 823.79 *ms*, enquanto o Waves necessita de 827.64 *ms* para processar essas consultas. Para o cenário top-1000, houve 21 consultas que necessitaram de três waves. Nesse caso, os métodos BMWT e MBMWT foram melhores do que os métodos BMW-CSP e Waves.

Um comentário sobre as consultas que vão para a terceira wave é que todas elas são longas. Além disso, como podemos ver nos experimentos, são menos de 0,3% das consultas que requerem a terceira wave. Notamos que, mesmo nessas consultas, que representam a pior situação para o método Waves, o algoritmo ainda alcança desempenho de tempo competitivo quando comparado aos *baselines* considerados.

	Tempo (ms)		
#waves	1	2	3
top-10			
#queries	2827	7145	28
BMWT	$7,34 \pm 0,95$	$57,59 \pm 2,04$	$1,125,03 \pm 317,87$
MBMWT	$6,03 \pm 0,90$	$42,95 \pm 1,72$	$1,063,86 \pm 340,09$
BMW-CSP	$3,09 \pm 0,35$	$23,11 \pm 0,77$	**823,79** $\pm 329,60$
Waves	**1,59** $\pm 0,18$	**18,58** $\pm 0,80$	$827,64 \pm 321,93$
top-1000			
#queries	1610	8369	21
BMWT	$12,15 \pm 0,95$	$123,02 \pm 3,31$	$1,788,44 \pm 211,02$
MBMWT	$9,57 \pm 0,75$	$92,99 \pm 2,57$	**1,688,39** $\pm 221,60$
BMW-CSP	$8,19 \pm 0,56$	$72,88 \pm 1,77$	$2,042,86 \pm 242,66$
Waves	**4,40** $\pm 0,36$	**62,99** $\pm 1,94$	$1,892,60 \pm 248,20$

Tabela 5.10: Tempo médio dos métodos com intervalo de confiança de 95% durante o processamento de consultas agrupadas de acordo com o número de waves exigidas pelo método Waves.

5.5 Aplicando técnicas de reatribuição de identificadores aos documentos

Uma estratégia importante para otimizar ainda mais os métodos de processamento de consulta é aplicar técnicas de reatribuição de identificadores de documentos, utilizando estratégias de agrupamento na tentativa de atribuir identificadores semelhantes aos documentos semelhantes. Enquanto a maioria das técnicas de reatribuição de identificadores é baseada em estratégias de agrupamento computacionalmente caras, uma técnica simples e barata foi proposta por Silvestri [30] para o caso específico das coleções da Web. Ele classifica os documentos pela ordem lexicográfica de suas URLs e usa a ordenação obtida para reatribuir os identificadores dos documentos. Os autores dos *baselines* BMW e MBMW também realizaram experimentos com uso dessa mesma técnica de reatribuição de identificadores no trabalho referenciado em [29].

Realizar experimentos com reatribuição de identificadores é importante para evitar dúvidas sobre as possíveis diferenças de desempenho entre o método BMW e os métodos multi-camadas propostos. Podem surgir dúvidas sobre a possibilidade das melhorias alcançadas por métodos multi-camadas serem apenas pelo fato de que eles modificam a ordenação dos documentos ao dividir as listas em níveis distintos. Entretanto, a aplicação de técnicas de reatribuição de identificadores de documentos leva a melhorias de desempenho tanto no processamento de consultas com o método BMW quanto no processamento

com os métodos multi-camadas.

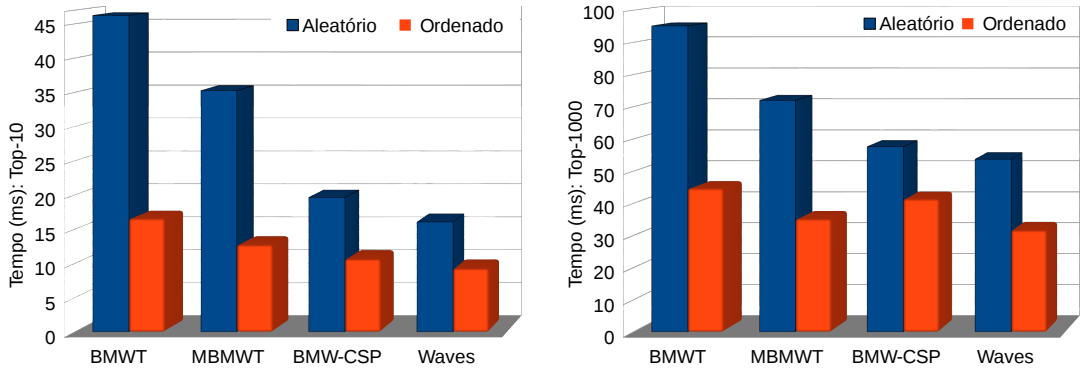

Figura 5.12: Tempos alcançados pelos métodos BMWT, MBMWT, BMW-CSP e Waves ao utilizar a técnica de reatribuição de identificadores de documentos para indexar a coleção GOV2, em comparação com os tempos obtidos quando se utiliza uma atribuição de ID de documento aleatório.

As diferenças de desempenho alcançadas com cada método adotando essa técnica são apresentadas nos gráficos da Figura 5.12. O tempo médio de processamento do método BMW passou de $46,21$ *ms* para $16,55$ *ms* ao computar top-10, e de $95,04$ *ms* para $44,58$ *ms* para computar top-1000. Esse primeiro resultado confirma a importância da reatribuição de identificadores como uma técnica para reduzir o tempo de processamento de consultas.

No método Waves, a reatribuição de identificadores reduziu o tempo de processamento de $16,08$ *ms* para $9,24$ *ms* e de $53,71$ *ms* para $31,57$ *ms*, ao computar top-10 e top-1000, respectivamente. Quando comparado com o tempo utilizado pelo BMW, o método Waves reduziu os tempos de processamento de consultas em cerca de $44,2\%$ e $29,2\%$, respectivamente para top-10 e top-1000 respostas. Os resultados mostram que o método Waves tem o melhor desempenho, mesmo nesse cenário.

5.6 ClueWeb09

Essa seção considera a importância de verificar se as diferenças de desempenho entre os métodos avaliados também ocorrem na execução de experimentos com outra base de dados, especialmente para coleções de documentos maiores como a ClueWeb09. Os resultados alcançados com cada método na nova coleção são apresentados na Tabela 5.11. Além do tempo médio do processamento de todo o conjunto de consultas, a tabela contém o tempo médio de processamento ao agrupar as consultas por número de termos. As

diferenças de desempenho ao comparar os métodos são aproximadamente as mesmas que são obtidas durante a realização de experimentos com a coleção GOV2. O método BMW-CSP é 28% e 9% mais rápido que o método MBMWT respectivamente para retornar os rankings top-10 e top-1000. O método Waves chega a ser aproximadamente 61% mais rápido para computar o ranking top-10 e aproximadamente 46% mais rápido ao computar o ranking top-1000 do que o MBMWT.

	#Bloco	#Pivô	Tempo (ms)	1	2	3	4	5	>5
	top-10								
BMWT	15.482	181.891	$13,06 \pm 2,41$	0,74	4,48	14,23	31,20	57,67	200,81
MBMWT	13.719	122.501	$10,97 \pm 1,92$	0,77	4,41	12,41	25,90	44,21	148,19
BMW-CSP	5.715	95.688	$7,85 \pm 0,75$	0,31	3,56	9,57	16,79	25,93	56,17
Waves	3.779	76.468	$\mathbf{4,27} \pm 0,59$	**0,07**	**1,64**	**5,05**	**10,34**	**15,88**	**42,50**
	top-1000								
BMWT	42.994	683.119	$44,62 \pm 5,32$	3,15	17,90	51,69	109,34	174,97	534,12
MBMWT	32.472	418.113	$33,36 \pm 3,96$	2,60	14,36	39,41	81,68	123,18	356,56
BMW-CSP	14.883	480.486	$30,27 \pm 2,19$	2,82	18,12	39,41	63,93	84,24	190,38
Waves	14.580	188.081	$\mathbf{18,22} \pm 1,61$	**0,96**	**7,89**	**23,04**	**45,68**	**66,39**	**137,38**

Tabela 5.11: Número médio de blocos acessados, número médio de pivôs avaliados e desempenho médio de tempo com intervalo de confiança de 95%, para processar todas as consultas e para diferentes tamanhos de consultas, ao computar top-10 e top-1000 resultados na base de dados ClueWeb09.

Os resultados dos experimentos ao considerar o tamanho das consultas mostram que o método BMW-CSP é mais rápido que os *baselines* para qualquer tamanho de consulta ao computar as top-10 respostas. Ao computar as top-1000 respostas, o método BMW-CSP é mais rápido que o melhor *baseline* nos quatro grupos compostos por consultas com mais de 2 termos. Com o método Waves o tempo necessário para processar os mesmos grupos de consultas é menor para qualquer um dos grupos tanto ao retornar os top-10 resultados quanto para retornar os top-1000 resultados.

5.7 Índice Comprimido

Os experimentos reportados até a seção anterior não incluem métodos de compressão para reduzir o espaço ocupado pelas listas invertidas. Como normalmente as listas invertidas são muito grandes, é comum que se empreguem métodos de compressão sobre as entradas armazenadas nessas listas.

Comprimir sequências de inteiros, como as listas invertidas, é um problema que tem sido muito estudado. A literatura apresenta várias abordagens, cada qual introduz seu

próprio *trade-off* entre espaço ocupado e velocidade de descompressão [16, 19, 26, 31, 34].

Catena et. al [10] analisam e comparam o desempenho de diferentes métodos de compressão encontrados na literatura, como: Simple16, FOR, PForDelta, NewPFD, OptPFD e FastPFOR. Para os experimentos adotados neste trabalho adotamos o método FastPFOR.

Em listas ordenadas por documento, ao invés de armazenar o valor do *docId*, costuma-se armazenar apenas a diferença entre o *docId* do documento atual na lista e o seu anterior. Isso reduz os valores inteiros a serem comprimidos, melhorando a taxa de compressão e consequentemente reduzindo o tamanho físico dos índices utilizados no processamento de consultas.

Ao aplicar um método de compressão é acrescentado ao processamento o custo de descomprimir informações da lista sempre que um novo bloco for acessado. Esse custo extra na descompressão poderia aumentar a distância entre os tempos de processamento dos *baselines* e os novos métodos propostos, dado que os novos métodos reduzem consideravelmente o número de acesso a blocos de documentos nas listas invertidas.

O impacto prático da aplicação de técnicas de compressão de dados a sistemas de busca que utilizem cada um dos métodos aqui estudados são apresentados na Tabela 5.12.

	#Bloco	#Pivô	Tempo (ms)
	top-10		
BMWT	15.482	181.891	$13,24 \pm 2,41$
MBMWT	13.719	122.501	$10,94 \pm 1,92$
BMW-CSP	5.715	95.688	$8,06 \pm 0,75$
Waves	3.779	76.468	$4,04 \pm 0,59$
	top-1000		
BMWT	42.994	683.119	$44,60 \pm 5,32$
MBMWT	32.472	418.113	$33,31 \pm 3,96$
BMW-CSP	14.883	480.486	$31,30 \pm 2,19$
Waves	14.580	188.081	$18,11 \pm 1,61$

Tabela 5.12: Número médio de blocos descomprimidos, número médio de pivôs avaliados e desempenho médio de tempo com intervalo de confiança de 95% ao computar top-10 e top-1000 resultados na base de dados ClueWeb09 utilizando o método de compressão FastPFOR.

Os resultados apresentados na Tabela 5.12 correspondem ao desempenho médio em tempo de processamento de cada método ao computar top-10 e top-1000 resultados na base de dados ClueWeb09 utilizando o método de compressão FastPFOR. Com o método de compressão alcançamos uma grande redução no espaço ocupado pelo índice sem afetar

o tempo de processamento. O índice invertido descomprimido ocupa aproximadamente 142Gb enquanto que o mesmo índice comprimido com o método FastPFOR ocupa apenas 19% desse espaço, 27*Gb*.

Capítulo 6

Conclusões

Nesta tese apresentamos e avaliamos dois novos métodos de processamento de consultas, o método BMW-CSP e o método Waves, que foram propostos com o objetivo de reduzir o tempo de processamento de consultas utilizando técnicas que reduzem o número de documentos computados. Foram implementados, para avaliação, os métodos WAND, BMW, MBMW, BMW-CS, BMW-CSP e Waves. Identificamos o melhor ajuste de cada método para garantir uma comparação justa em todos os distintos cenários de experimentos adotados.

O primeiro método proposto nesta tese, chamado BMW-CSP [11], modifica o método BMW-CS proposto por Rossi et al.[25] para garantir o processamento de consultas sem a perda de resultados ocorrida no BMW-CS. O resultado dos experimentos demonstram que o método BMW-CSP chega a alcançar uma redução no tempo de processamento de aproximadamente 47% para retornar as top-10 respostas e de 20% para retornar as top-1000 respostas quando comparado ao melhor *baseline*, MBMW. O segundo método, Waves, tem como vantagem dispensar o espaço extra para armazenar resultados parciais que é necessário nos métodos BMW-CS e BMW-CSP. Nossos experimentos demostraram que o método Waves, em comparação a outros métodos, conseguiu explorar melhor a possibilidade de divisão do índice em camadas, o que resultou em uma redução de cerca de 57% do tempo de processamento para cenário que retorna as top-10 respostas e uma redução de cerca de $30,5\%$ do tempo de processamento no cenário que retorna top-1000 respostas em relação ao método MBMW.

Para trabalhos futuros, sugerimos um estudo sobre como estimar de forma analítica o melhor tamanho e número de camadas para cada método, buscando assim dispensar a

necessidade da realização de experimentos prévios de teste para essa definição. Ainda que o custo de execução de testes para ajustar os parâmetros não seja proibitivo, um estudo para eliminar tais ajustes seria interessante.

Outra sugestão é o estudo do impacto da utilização de nossos métodos em sistemas onde o índice completo não cabe em memória, verificando se a estratégia multi-camadas aqui proposta seria adequada a esse cenário. Quando temos vários níveis de memória em hierarquia, as estratégias multi-camadas podem competir ou complementar o desempenho de sistemas de cache. Normalmente técnicas de cache são usadas nos processadores de consulta em casos onde o índice invertido não cabe em memória. Nesses casos, durante o processamento de consultas, o disco é acessado para carregar parte dos índices que estão sendo processados. Basicamente, o objetivo de técnicas de cache é reduzir o número de acessos ao disco durante o processamento de consultas, mantendo em memória blocos dos índices que são comumente acessados. Todos os experimentos apresentados neste trabalho foram realizados com o índice em memória. Apesar de demonstrarmos que os métodos BMW-CSP e Waves reduzem o tempo de processamento de consultas por reduzirem o número de acessos ao índice, uma sugestão de trabalhos futuros seria avaliar o desempenho dos métodos em um ambiente onde o índice não caiba em memória e aplicar técnicas de cache para gerenciar o acesso ao disco.

Por fim, uma última sugestão para trabalhos futuros é a avaliação dos algoritmos com diferentes funções de similaridade. Em todos os *baselines* e novos métodos apresentados aqui, a função de similaridade pode ser determinante para o desempenho das técnicas de poda adotadas. Como pode ser visto no Capítulo 3, nos métodos que usam índice em camadas, a distribuição em camadas é feita com base no peso dos documentos na lista invertida. Todos os limiares usados para o processo de saltar documentos, como o *Max Score*, *Min Score*, *Block Max Score* e o limiar de descarte, também são gerados a partir da função de similaridade. Visto que a função de similaridade adotada pode influenciar o tempo de resposta dos algoritmos propostos, é interessante que se realizem experimentos para avaliar o impacto de tais funções nos diversos algoritmos estudados.

Bibliografia

[1] ANAGNOSTOPOULOS, A., BECCHETTI, L., LEONARDI, S., MELE, I., AND SANKOWSKI, P. Stochastic query covering. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining* (New York, NY, USA, 2011), WSDM '11, ACM, pp. 725–734.

[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* (New York, NY, USA, 2006), SIGIR '06, ACM, pp. 372–379.

[3] BAEZA-YATES, R., GIONIS, A., JUNQUEIRA, F., MURDOCK, V., PLACHOURAS, V., AND SILVESTRI, F. The impact of caching on search engines. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (New York, NY, USA, 2007), SIGIR '07, ACM, pp. 183–190.

[4] BAEZA-YATES, R., GIONIS, A., JUNQUEIRA, F. P., MURDOCK, V., PLACHOURAS, V., AND SILVESTRI, F. Design trade-offs for search engine caching. *ACM Transactions on the Web 2*, 4 (Oct. 2008), 20–28.

[5] BAEZA-YATES, R., MURDOCK, V., AND HAUFF, C. Efficiency trade-offs in two-tier web search systems. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval* (New York, NY, USA, 2009), SIGIR '09, ACM, pp. 163–170.

[6] BAEZA-YATES, R., AND RIBEIRO-NETO, B. Modern information retrieval: the concepts and technology behind search. harlow (reino unido), 2011.

[7] BLANCO, R., AND BARREIRO, A. *Static Pruning of Terms in Inverted Files*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 64–75.

[8] 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 12th International Conference on Information and Knowledge Management* (New York, NY, USA, 2003), CIKM '03, ACM, pp. 426–434.

[9] 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* (New York, NY, USA, 2001), SIGIR '01, ACM, pp. 43–50.

[10] CATENA, M., MACDONALD, C., AND OUNIS, I. *On Inverted Index Compression for Search Engine Efficiency*. Springer International Publishing, Cham, 2014, pp. 359–371.

[11] DAOUD, C. M., DE MOURA, E. S., CARVALHO, A., DA SILVA, A. S., FERNANDES, D., AND ROSSI, C. Fast top-k preserving query processing using two-tier indexes. *Information Processing & Management 52*, 5 (2016), 855 – 872.

[12] DE CARVALHO, L. L. S., DE MOURA, E. S., DAOUD, C. M., AND DA SILVA, A. S. Heurísticas para aprimorar o método BMW e suas variantes. In *XXX Simpósio Brasileiro de Banco de Dados - Short Papers, Petrópolis, Rio de Janeiro, Brasil, October 13-16, 2015.* (2015), pp. 113–124.

[13] 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* (New York, NY, USA, 2005), WWW '05, ACM, pp. 235–244.

[14] DING, S., AND SUEL, T. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th International ACM SIGIR Conference on Rese-*

arch and Development in Information Retrieval (New York, NY, USA, 2011), SIGIR '11, ACM, pp. 993–1002.

[15] GROSSMAN, D. A., AND FRIEDER, O. *Information retrieval: Algorithms and heuristics*, vol. 15. Springer Science & Business Media, 2012.

[16] LEMIRE, D., AND BOYTSOV, L. Decoding billions of integers per second through vectorization. *Software: Practice and Experience 45*, 1 (Jan. 2015), 1–29.

[17] LIU, Y., ZHANG, M., CEN, R., RU, L., AND MA, S. Data cleansing for web information retrieval using query independent features. *Journal of the American Society for Information Science and Technology 58*, 12 (2007), 1884–1898.

[18] LONG, X., AND SUEL, T. Three-level caching for efficient query processing in large web search engines. In *Proceedings of the 14th International Conference on World Wide Web* (New York, NY, USA, 2005), WWW '05, ACM, pp. 257–266.

[19] MOFFAT, A., AND STUIVER, L. Binary interpolative coding for effective index compression. *Information Retrieval 3*, 1 (2000), 25–47.

[20] NGUYEN, L. T. Static index pruning for information retrieval systems: A posting-based approach. In *SIGIR 2009 Workshop on Large-Scale Distributed Information Retrieval* (2009), pp. 25–32.

[21] NTOULAS, A., AND CHO, J. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (New York, NY, USA, 2007), SIGIR '07, ACM, pp. 191–198.

[22] PERSIN, M., ZOBEL, J., AND SACKS-DAVIS, R. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science and Technology 47*, 10 (1996), 749–764.

[23] RISVIK, K. M., AASHEIM, Y., AND LIDAL, M. Multi-tier architecture for web search engines. In *Proceedings of the First Conference on Latin American Web Congress* (Washington, DC, USA, 2003), LA-WEB '03, IEEE Computer Society, pp. 132–143.

[24] 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* (New York, NY, USA, 1994), SIGIR '94, Springer-Verlag New York, Inc., pp. 232–241.

[25] 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* (New York, NY, USA, 2013), SIGIR '13, ACM, pp. 183–192.

[26] SALOMON, D. *Variable-length codes for data compression*. Springer Science & Business Media, 2007.

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

[28] SARAIVA, P. C., SILVA DE MOURA, E., ZIVIANI, N., MEIRA, W., FONSECA, R., AND RIBERIO-NETO, B. Rank-preserving two-level caching for scalable search engines. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval* (2001), ACM, pp. 51–58.

[29] SHAN, D., DING, S., HE, J., YAN, H., AND LI, X. Optimized top-k processing with global page scores on block-max indexes. In *Proceedings of the 5th ACM International Conference on Web Search and Data Mining* (New York, NY, USA, 2012), WSDM '12, ACM, pp. 423–432.

[30] SILVESTRI, F. Sorting out the document identifier assignment problem. In *Advances in Information Retrieval: 29th European Conference on IR Research, ECIR 2007, Rome, Italy, April 2-5, 2007. Proceedings* (Berlin, Heidelberg, 2007), G. Amati, C. Carpineto, and G. Romano, Eds., Springer Berlin Heidelberg, pp. 101–112.

[31] STEPANOV, A. A., GANGOLLI, A. R., ROSE, D. E., ERNST, R. J., AND OBEROI, P. S. Simd-based decoding of posting lists. In *Proceedings of the 20th ACM*

International Conference on Information and Knowledge Management (New York, NY, USA, 2011), CIKM '11, ACM, pp. 317–326.

[32] 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* (New York, NY, USA, 2007), SIGIR '07, ACM, pp. 175–182.

[33] THEOBALD, M., WEIKUM, G., AND SCHENKEL, R. Top-k query evaluation with probabilistic guarantees. In *Proceedings of the 30th International Conference on Very Large Data Bases* (2004), VLDB'04, VLDB Endowment, pp. 648–659.

[34] 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* (New York, NY, USA, 2009), WWW'09, ACM, pp. 401–410.

[35] 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, IEEE, pp. 697–702.