PROCESSAMENTO DE CONSULTAS DOCUMENTO-A-DOCUMENTO UTILIZANDO ÍNDICE EM CAMADAS

CRISTIAN ROSSI

PROCESSAMENTO DE CONSULTAS DOCUMENTO-A-DOCUMENTO UTILIZANDO ÍNDICE EM CAMADAS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Federal do Amazonas como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Edleno Silva de Moura

Manaus - AM

Março de 2013

Agradecimentos

Agradeço a Deus por ter me iluminado durante esta caminhada. Agradeço a minha mãe Elizabeth Paranhos Rossi, e a meu pai Vilmar Rossi, pelo suporte durante todos estes anos de aprendizados e crescimento, pelo imenso apoio aos estudos durante a infância e pelos ensinamentos de caráter, ética e respeito ao próximo. Agradeço a minhas irmãs, Maiane Rossi e Amanda Paranhos Rossi, por serem parte da minha vida e por todos os momentos que passamos juntos. Agradeço a minha amada namorada, Ludimila Carvalho Gonçalves, por me acompanhar durante esta caminhada e pelo apoio, carinho e paciência durante as mais diversas dificuldades. Agradeço aos meus amigos por todos os momentos de parceria durante esta caminhada, pelos momentos de alegria e alívio, e pelo imenso apoio recebido. Agradeço aos meus professores, e principalmente ao meu orientador Edleno Silva de Moura, por todos os ensinamentos que levarei para o resto da minha vida profissional. Por fim, agradeço a Universidade Federal do Amazonas pela oportunidade de aprendizado e crescimento, e a Fundação de Amparo à Pesquisa do Estado do Amazonas pelo suporte financeiro concedido.

"Amai-vos uns aos outros como eu vos amei."
(Jesus)

Resumo

Sistemas de busca são mecanismos capazes de buscar informação relevante dentro de grandes coleções de dados. O constante crescimento de meios eletrônicos para armazenamento de informação, junto com a popularização dos sistemas de busca, traz consigo a necessidade constante por soluções capazes de reduzir os custos de processamento de consultas.

Neste trabalho, apresentamos dois novos algoritmos para processamento de consultas em sistemas de busca. Os algoritmos utilizam a abordagem de processamento documento-a-documento e modificam o atual algoritmo *estado-da-arte*, BMW, para tirar vantagem de uma arquitetura de índice dividido em duas camadas. A primeira camada contém apenas as entradas de maior impacto do índice e é utilizada para preprocessar as consultas antes de acessar o restante do índice na segunda camada. Esta abordagem resulta em consideráveis ganhos de desempenho.

O primeiro algoritmo proposto, chamado BMW-CS, chega a ser 40 vezes mais rápido em relação a diversos métodos comparados, porém provoca pequenas modificações no conjunto de resposta retornado. O segundo algoritmo proposto, chamado BMW-t, preserva o conjunto de resposta e é 10% mais rápido que o BMW.

Palavras-chave: Recuperação de Informação, Processamento de Consultas, Índices Invertidos, Sistemas de Busca.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Sistemas de busca são mecanismos capazes de procurar informação relevante dentro de uma coleção de documentos de acordo com as necessidades do usuário. Nesses sistemas, é comum o usuário especificar suas necessidades de informação por meio de uma consulta, geralmente descrita por um conjunto pequeno de palavras-chave. A partir deste conjunto de palavras-chave, também chamadas de termos, o sistema vasculha a coleção em busca de documentos que possam ser relevantes para o usuário. Ao final do processo, o sistema retorna um conjunto de documentos ordenados por algum valor que representa uma estimativa de relevância.

O processador de consultas é uma parte primordial de qualquer sistema de busca, pois é o responsável por atender às consultas especificadas pelos usuários. O resultado final para uma dada consulta, em um sistema de busca, utiliza em geral um modelo de recuperação de informação (RI), como BM25 [Robertson & Walker, 1994] ou o Modelo de Espaço Vetorial [Salton et al., 1974], para calcular uma pontuação que é utilizada para ordenar os resultados apresentados aos seus usuários. Os documentos com melhor pontuação são selecionados para compor o resultado (também chamado de top-k, onde k é o número de documentos retornados) que será retornado para o usuário ou será utilizado por outro processo de ordenação mais complexo [Carvalho et al., 2012], que leva em consideração novas fontes de evidência de relevância para calcular o resultado final. Mesmo nessa última hipótese, o processamento dos documentos para que se obtenha a ordenacão inicial de resultados com um modelo de RI é parte determinante no custo final do processamento de consultas em um sistema de busca.

O constante crescimento de meios eletrônicos para armazenamento de informação, junto com a popularização de sistemas de busca disponíveis para encontrar informação relevante armazenada nestes meios, traz consigo a necessidade constante por soluções que possam reduzir os custos para processamento de consultas. O tamanho das cole-

ções de documentos processadas por sistemas de busca está constantemente crescendo, conforme mais informação é disponibilizada em meios digitais. Tal crescimento não é acompanhado pela evolução do poder de processamento das máquinas atuais.

Apesar de índices utilizados por sistemas de busca proverem um acesso rápido aos documentos onde os termos de uma consulta ocorrem, eles crescem de acordo com o tamanho das coleções de documentos. O custo de se processar consultas cresce linearmente conforme o tamanho da coleção. Para diminuir este 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.

De acordo com Jansen & Spink [2003], apesar de um sistema de busca indexar uma grande quantidade de documentos, aproximadamente 80% dos usuários analisam, no máximo, os trinta primeiros resultados dados como resposta a uma consulta. O foco do usuário sobre os primeiros resultados levou ao desenvolvimento de várias técnicas de processamento visando recuperar apenas um pequeno conjunto de respostas com os top-k documentos mais relevantes, onde k é um valor pequeno, tipicamente menor que 1000, de forma que o custo de processamento seja menor.

Neste trabalho, apresentamos dois novos algoritmos para computar os top-k documentos de maior pontuação para uma consulta. Os algoritmos apresentados tiram vantagem de um índice em duas camadas, onde a primeira camada contém informação que é usada para aumentar a velocidade de processamento obtida ao se processar a segunda camada. O primeiro algoritmo proposto BMW-CS, computa as respostas de maior pontuação de maneira aproximada, podendo eventualmente modificar os resultados em relação ao que seria obtido pelo sistema de busca sem o uso de nosso algoritmo. O segundo algoritmo, chamado BMW-t, preserva os resultados do sistema.

Os algoritmos desenvolvidos modificam o melhor *baseline* que encontramos na literatura, o *Block-Max WAND* (*BMW*) proposto por Ding & Suel [2011], para tirar vantagem da arquitetura em duas camadas estudada. Nossos resultados mostram que o *BMW-CS* atinge grandes ganhos de velocidade, sendo várias vezes mais rápido que o BMW, porém não garante que o conjunto exato de respostas será retornado. O *BMW-t*, apesar de mais lento que o BMW-CS, supera o *baseline* BMW, e garante que o conjunto exato de respostas será retornado.

1.1 Organização do Trabalho

O trabalho está organizado como se segue. A seguir, no Capítulo 2, apresentamos os conceitos básicos para o entendimento de sistemas de busca e processamento de

consultas. No Capítulo 3 mostramos um resumo da literatura sobre métodos de processamento eficiente de consultas. No Capítulo 4 apresentamos os algoritmos para processamento de consultas nos quais os algoritmos desenvolvidos neste trabalho foram baseados. No Capítulo 5, detalhamos os algoritmos desenvolvidos BMW-CS e BMW-t. No Capítulo 6, detalhamos o ambiente de experimentação utilizado neste trabalho para avaliarmos os algoritmos desenvolvidos e discutimos os resultados atingidos. Finalmente, no Capítulo 7, fazemos a conclusão do trabalho e apontamos algumas direções para trabalhos futuros.

Capítulo 2

Conceitos Básicos

Neste capítulo, apresentamos um conjunto de conceitos para o melhor entendimento do trabalho apresentado.

2.1 Sistemas de Busca

Em um sistema de busca, um documento é descrito principalmente por seus termos. O conjunto de termos distintos dentro da coleção de documentos é chamado *vocabulário*. Quando um usuário especifica uma consulta por meio de um conjunto de termos, o sistema de busca encontra os top-k documentos mais similares com essa consulta, que contenham os termos da consulta.

Para acelerar a busca por documentos que contenham os termos da consulta, é comum empregar a estrutura de *índices invertidos* [Baeza-Yates & Ribeiro-Neto, 2011]. 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 no documento. Cada termo na coleção recebe um identificador numérico, assim como os documentos da coleção. Neste trabalho chamaremos o identificador numérico de um termo como sendo $termId$, e do documento como $docId$. Cada entrada em uma lista invertida corresponde a um par $(docId, freq)$, onde $docId$ é o identificador do documento e $freq$ a frequência do termo no documento.

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 término do processo de indexação, as informações geradas são disponibilizadas para que o sistema de busca utilize-as no processamento de consultas.

Este 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 destas 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). Esse processo é conhecido como *ranking*.

2.2 Similaridade

Há vários trabalhos na literatura que propõem fórmulas para o cálculo da similaridade entre um documento e uma consulta. Um dos modelos mais tradicionais na área de RI é o Modelo de Espaço Vetorial proposto por Salton et al. [1974], onde os documentos são representados como vetores, e o vocabulário é a dimensão dos vetores. Dois documentos tem seu valor de similaridade determinado pela formula do cosseno, onde documentos idênticos tem valor de similaridade 1 e documentos completamente diferentes similaridade 0.

Outro modelo de similaridade popular na literatura é o BM25 [Robertson & Walker, 1994]. Este é baseado em um modelo probabilístico e tem sido amplamente utilizado por sistemas de busca. Todos os algoritmos de processamento de consultas estudados neste trabalho utilizaram esse modelo para o cálculo da similaridade. A seguir temos a fórmula do BM25:

$$BM25(D,Q) = \sum_{i=1}^{n} IDF(q_i) * \frac{f(q_i,D) * (k_1+1)}{f(q_i,D) + k_1 * (1 - b + b * \frac{|D|}{avgdl})} \tag{2.1}$$

Onde

$$IDF(q_i) = log(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5}) \tag{2.2}$$

Na Equação 2.2, $IDF(q_i)$ representa a frequência inversa do termo q_i na coleção. Onde N é o número total de documentos dentro da coleção e $n(q_i)$ representa o número de documentos da coleção onde o termo q_i ocorreu. O valor de *idf* de um termo busca representar a importância do termo dentro da coleção. Termos muito frequêntes tem valores de *idf* baixo, enquanto termos mais raros tem valor de *idf* mais alto, indicando que o termo é mais importante.

Na Equação 2.1, temos a função de similaridade entre um documento D e uma consulta Q. $f(q_i, D)$ representa a frequência do termo q_i no documento D. $|D|$ representa a norma, ou "tamanho", do documento D, e $avgdl$ representa a norma média dos documentos da coleção. Em nossos experimentos, definimos os parâmetros $b = 0.75$ e

$k_1 = 2$. A norma de um documento corresponde ao número de termos distintos dentro do documento.

Ao longo do trabalho, chamaremos o valor de similaridade também de *score*.

2.3 Organização do Índice invertido

A organização de um índice invertido é determinante na escolha de qual algoritmo será utilizado no processamento de consultas. Como as listas invertidas são grandes, é comum empregar métodos de compressão sobre as entradas armazenadas em uma lista invertida. 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.

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 $docId$. Trabalhos recentes propuseram que ao invés de se armazenar a frequência, os índices invertidos pudessem armazenar diretamente o valor de similaridade do termo com o documento [Anh & Moffat, 2006] pré-computado dentro da lista invertida. 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 identificador. 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.

Uma estrutura empregada em índices invertidos ordenados por documento para acelerar o acesso aleatório sobre as entradas das listas invertidas é a estrutura de *skiplists*. Para cada lista invertida é gerado uma *skiplist*, armazenando 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 descomprimi-las.

Neste trabalho optamos pela utilização de índices ordenados por documento. To-

Listing 2.1. Algoritmo Genérico para Processamento de consultas

```
1  QueryProcessing(Q, k, Collection C)
2    A ← {}
3
4    for each document D ∈ C
5      score ← sim(D, Q)
6      A ← A∪ (D, score)
7    end for
8
9    sort A by score
10
11   return A[0..k]
```

dos os algoritmos implementados trabalham com índices ordenados por documento. Cada lista invertida possui uma *skiplist* associada, contendo uma entrada para cada bloco de 128 documentos.

2.4 Processamento de consultas

Na Listagem 2.1, temos o pseudo-código com a descrição de um algoritmo genérico para processamento de consultas, onde Q representa a consulta submetida para o sistema, k é o número de resultados solicitados, e C é a coleção de documentos. O sistema calcula a similaridade de cada documento da coleção com a consulta e armazena no conjunto de acumuladores $\mathcal{A}$. Ao final do processo é retornado os top-k documentos de maior similaridade com a consulta.

Como dito anteriormente, a organização do índice invertido determina qual algoritmo será melhor aplicado para o processamento de consultas. Seguindo as formas de organização de um índice invertido, temos dois conjuntos principais de algoritmos para processar consultas: i)Processamento termo-a-termo (TAT), que utiliza índices ordenados por frequência. ii)Processamento Documento-a-documento (DAD), 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 Persin et al. [1996], onde o processamento é interrompido quando se tem certeza 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, assim, 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 contém 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 nesta mudança no conjunto de resposta.

Atualmente o algoritmo de processamento de consultas termo-a-termo mais eficiente foi desenvolvido por Strohman & Croft [2007], onde as listas são percorridas em paralelo, utilizando um índice ordenado por *score*. Esse método foi superado pelo método de processamento documento-a-documento BMW Ding & Suel [2011], o qual será utilizado como *baseline* para compararmos o desempenho dos algoritmos propostos neste trabalho.

Capítulo 3

Trabalhos Relacionados

Neste capítulo apresentamos um levantamento bibliográfico sobre as técnicas utilizadas por sistemas de busca para melhorar o desempenho no processamento de consultas e a capacidade de tratar grandes massas de dados.

3.1 Cache

Segundo Baeza-Yates et al. [2008], técnicas de *cache* assumem que existe uma taxa de repetição na sequência de entradas que são submetidas ao sistema durante um período de tempo, possibilitando que possa ser feito um *cache* com taxas de acerto efetivas, mesmo utilizando-se uma quantidade limitada de memória para tanto. Em sistemas de busca, a distribuição de consultas submetidas e os termos utilizados mudam lentamente ao longo do tempo. Esta característica possibilita que técnicas de *cache* sejam aplicadas para diminuir o tempo de processamento de consultas.

Cache pode ser aplicado em diversos níveis do processamento de consulta. Saraiva & de Moura [2001] 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 processamento em uma máquina de busca real. Os autores mostram que a utilização dessas duas estruturas ajuda a aumentar o a quantidade de consultas simultaneas processadas em relação a uma arquitetura sem um esquema de *cache*.

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. Baeza-Yates et al. [2008], através da análise de um histórico de consultas do Yahoo!, mostra 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%. Enquanto o *cache* de termos é limitado pela fração de termos distintos presentes nas consultas feitas, ficando com as taxas de acerto em torno de 90%.

As estratégias para preenchimento do *cache* podem ser divididas entre estática e dinâmica. Um *cache* estático seleciona um conjunto de itens que é refeito periodicamente, podendo ser composto de respostas ou listas invertidas. A escolha das entradas que serão armazenadas no *cache* é baseada em informações históricas, como por exemplo a lista consultas submetidas durante um período de tempo. Em *caches* dinâmicos, os itens mudam de acordo com a sua utilização. Geralmente aplica-se uma política de substituição de itens para definir quem deve ser retirado quando o *cache* está cheio, onde a LRU, substituição do menos recentemente utilizado, é a política mais adotada. O conteúdo de um *cache* dinâmico varia constantemente de acordo com a sequência de consultas feitas.

Em Baeza-Yates et al. [2007, 2008] é realizado um estudo comparativo entre opções de *cache* estático e dinâmico. 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 manter informação sobre a distribuição das consultas presentes no histórico, foi estudada a combinação de *cache* estático com dinâmico, dividindo a memória disponível para os dois. Verificou-se que uma mistura dos dois resulta em uma melhora nas taxas de acerto.

Long & Suel [2006] estudaram a utilização de um *cache* de interseção de listas invertidas. A interseção de pares de termos que ocorrem com muita frequência em históricos de consulta, é processada e armazenada em disco. Quando uma consulta possui três termos, contendo um par de termos com a sua interseção previamente calculada, ela é processada em conjunto com o outro termo, economizando o tempo que seria gasto para processar um 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. Porém em consultas com mais de dois termos, existe uma grande porcentagem de pares de termos que ocorrem juntos seguidos de outros termos.

Leonardi et al. [2011] aborda o problema de *cache* como um problema de cobertura de consultas, levando em consideração 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. [1996] criou 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.

Quando as entradas são removidas de acordo com os termos da consulta, o método de poda é classificado como poda dinâmica e 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 & Barreiro [2007] 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. Outros trabalhos, como Carmel et al. [2001], apresentam métodos de poda estática onde partes das entradas da lista são removidas. Este 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. [2005] apresenta uma variação do método de Carmel et al. [2001], 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. *Carmel* leva em consideração apenas informação individual dos termos do vocabulário, perdendo informação sobre termos que podem ocorrer em comum nas consultas. O método proposto por Moura et al. [2005] tenta solucionar esse problema analisando a co-ocorrência entre termos dentro de documentos da coleção. Além do topo da lista invertida de cada termo t, preserva-se também, para cada termo $t_2 \neq t$, 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.

Em 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 [2009] propôs um método que mistura poda baseada em termo e em documento, chamada poda baseada nas entradas das listas, com o objetivo de conciliar os defeitos de cada método, permitindo que uma entrada que teve um peso baixo em relação as demais entradas da lista invertida e obteve um peso alto em relação aos demais termos presentes no documento possa ser preservado durante a poda, e vice-versa.

Zheng & Cox [2009] desenvolveram um conjunto de estratégias para eliminar documentos da coleção baseando-se na premissa de que nem todos documentos são igualmente importantes. Apesar da idéia de remover documentos da coleção ser contra-intuitiva, 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.

Liu et al. [2007] aplicou aprendizado de máquina, para classificar documentos em uma coleção como de alta qualidade ou ordinários. Por meio desse método, é criado um índice contendo apenas documentos de alta qualidade. Nos experimentos realizados, foi possível 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 como bilhões de páginas e consultas variadas.

Anh & Moffat [2006] apresentam um método de poda dinâmica utilizando um índice ordenado por impacto. Estes 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.

Caso 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. Anh & Moffat [2006] apresenta outra versão do algoritmo, 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. Essa modificação resulta em ganhos de velocidade, porém o top-k de resposta não tem garantias de que será exato.

Strohman & Croft [2007] propuseram um novo método de processamento eficiente de consultas onde o índice é mantido em memória. O novo algoritmo modifica o método proposto por Anh & Moffat [2006]. 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 Broder et al. [2003], 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.

Broder et al. [2003] desenvolveu um novo algoritmo para processamento de consultas sobre índices ordenados por documento. O algoritmo utiliza um novo operador chamado WAND para percorre as listas invertidas dos termos de cada consulta em pa-

ralelo. Os documentos candidatos ao conjunto de resposta são identificados por meio de uma avaliação preliminar, considerando apenas informações dependentes da consulta. Quando um documento candidato é identificado, ele é avaliado completamente e seu valor de similaridade completo é computado. Quando um documento é completamente avaliado, devem-se levar em consideração informações relativas ao documento, como *Pagerank* e links. Isso causa um custo extra de processamento e transferência de informações do disco. O objetivo da fase de avaliação preliminar é possibilitar que grandes quantidades de entradas não sejam avaliadas completamente, garantindo assim uma economia de tempo. Na Seção 4.1 apresentaremos detalhes desse novo operador.

Ding & Suel [2011] propos uma modificação no método de Broder et al. [2003], chamada *Block-Max* WAND (BMW). O objetivo é aumentar a quantidade de entradas descartadas durante o processamento de consultas por meio da utilização de uma estrutura de índice que além de manter os limites de similaridade globais de cada lista invertida, mantém também os limites de similaridade dentro de cada bloco de entradas. As entradas das listas invertidas são agrupadas em blocos, que podem ser saltados sem a necessidade de descomprimir seu conteúdo. Cada bloco possui informações sobre o maior valor de similaridade de um documento encontrado dentro do bloco. Isto evita a leitura de blocos que não tenham candidatos ao conjunto de respostas.

Quando o iterador está percorrendo as listas, antes de se descomprimir as entradas em busca de documentos de resposta, é feita uma verificação se o bloco possui algum documento que possa ser candidato ao conjunto de respostas. Caso exista, as entradas do bloco são descomprimidas. Esse algoritmo é utilizado com *baseline* neste trabalho. Na Seção 4.2 apresentamos os algoritmo com mais detalhes.

Shan et al. [2012] mostra que o desempenho do BMW é degradado quando outras informações, como *Pagerank*, são adicionadas ao cálculo do *score* do documento. Eles então estudaram técnicas eficientes para processar o top-k em casos onde essas informação são incorporadas ao cálculo. Esse trabalho pode também ser aplicado a nossa proposta em um trabalho futuro, sendo ortogonal ao trabalho apresentado aqui.

3.3 Arquiteturas Multi-camadas

Um usuário que 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 preferem mudar a consulta. Sendo assim, o processamento de todos os documentos de uma lista invertida para uma dada consulta acarreta em um grande desperdício computa-

cional. Theobald et al. [2004] desenvolveu um método de processamento de consultas para computar os conjunto aproximado dos k documentos de maior similaridade com a consulta, de acordo com o modelo de RI adotado, sem a necessidade de processar todas 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.

De acordo com Baeza-Yates et al. [2009], o processamento em camadas pode ser feito de maneira sequencial. 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, que geralmente tem um pequeno conjunto de documentos, e por isso é mais rápida para ser processada.

Risvik et al. [2003] estudou a divisão do índice em camadas, de modo que a maioria das consultas pudessem ser processadas sem a necessidade de passar por todas as camadas do índice, diminuindo o tempo de processamento de uma consulta. No trabalho, é criada uma arquitetura de índice multi-camada, 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). Diferentemente de trabalhos anteriores, cada documento fica em uma única camada. O índice é dividido em 3 camadas, onde a primeira camada possui um pequeno conjunto de documentos, a segunda uma quantidade maior e a terceira o resto dos documentos. Graças à 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 (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 de Ntoulas & Cho [2007] 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

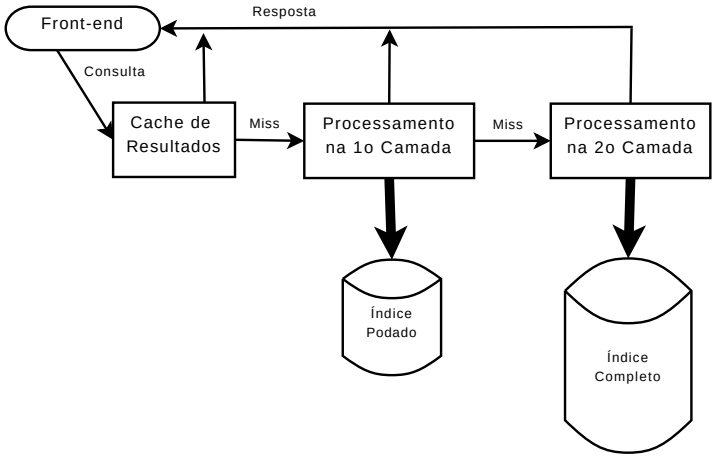

Figura 3.1. Arquitetura em duas Camadas utilizando um cache de resultados.

um algoritmo que determina se o conjunto de respostas retornado pelo primeiro índice é exatamente o mesmo que seria retornado pelo índice completo. Para isto 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. Skobeltsyn et al. [2008] 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. Conforme podemos observar na Figura 3.1, a arquitetura é passível de modificações, como a inserção de uma terceira camada para processamento, ou outra estrutura de *cache*. 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.

Baeza-Yates et al. [2009] estudou a utilização de um classificador SVM para determinar em qual índice uma consulta deve ser processada. O índice é dividido em duas camadas, sendo que a primeira camada contém um índice pequeno em relação ao índice da segunda. Caso o classificador escolha os primeira camada, a consulta é processada apenas sobre o índice desta camada. Caso o classificador erre, a penalidade é o tempo de processamento do segundo índice. Se o classificador indicar o segundo índice, o processamento é feito em paralelo nos dois índices, e caso o classificador erre, o processamento do segundo índice é suspenso quando o primeiro índice tiver termi-

nado, evitando um aumento no tempo de resposta, porém sobrecarregando o sistema que processa consultas feitas no segundo índice. Verificou-se que o desempenho do classificador tem impacto direto nos ganhos do sistema.

Capítulo 4

Processamento Documento-a-Documento

Neste trabalho, desenvolvemos algoritmos para processamento de consultas utilizando a abordagem documento-a-documento. Nesta alternativa de processamento, as lista invertidas de uma consulta, ordenadas por $docId$, são percorridas em paralelo, o que permite ao algoritmo computar o valor completo de similaridade para cada documento. Assim, é necessário armazenar apenas os top-k documentos de maior similaridade com a consulta, diminuindo a quantidade de memória necessária. Porém, como as listas invertidas estão ordenadas por $docId$, entradas importantes ficam espalhadas ao longo das listas, tornando a poda de documentos mais complexa.

Nas seções a seguir apresentamos a descrição de dois algoritmos de processamento DAD, sobre os quais os algoritmos desenvolvidos neste trabalho são baseados. Na Seção 4.1 apresentamos o algoritmo proposto por Broder et al. [2003] para processamento de consultas disjuntivas sobre índices ordenados por documento. Na Seção 4.2 apresentamos o algoritmo BMW, proposto por Ding & Suel [2011], o qual é baseado nas idéias propostas pelo algoritmo WAND e atualmente é o estado-da-arte para processamento de consultas disjuntivas sobre índices ordenados por documentos.

4.1 WAND

Broder et al. [2003] propôs um algoritmo de processamento, chamado WAND, que permite que o processamento de consultas conjuntivas e disjuntivas seja feito de modo eficiente sobre índices ordenados por documento. O WAND também pode ser configurado para preservar os top-k documentos de resposta ou não. Quando não se garante que o conjunto será preservado, o algoritmo atinge ganhos de velocidade maiores.

No WAND, as listas são percorridas de forma paralela. Um *heap* de acumuladores é criado para manter os top-k documentos de maior similaridade, onde k é o número de documentos de resposta requisitado. O menor valor de similaridade, chamado também de *score*, é utilizado como um *limiar de descarte* pelo algoritmo para acelerar o processamento. Um novo documento é avaliado e inserido no *heap* de acumuladores somente se tiver um *score* superior ao *limiar de descarte*.

A estrutura de dados *heap* é uma fila de prioridade mínima que permite acesso direto ao documento de menor *score* dentro do conjunto e permite que um novo documento seja inserido em tempo logarítmico.

O WAND tem dois níveis de avaliação. No primeiro nível, um documento pivô tem seu *score máximo possível* avaliado, utilizando a informação de *score* máximo de cada lista da consulta, onde o documento possa ter ocorrido. Se o *score máximo possível* for maior que o limiar de descarte atual, o documento tem seu *score* real avaliado. Caso contrário, o documento é descartado e um novo pivô é selecionado. Quanto mais documentos são descartados, menos entradas das listas invertidas são descomprimidas, e mais rápido é o processamento de uma consulta.

Para cada lista invertida, é calculado o maior valor de similaridade entre o termo e os documentos da lista, chamado *score máximo* da lista. Essa informação fica armazenada junto com as informações do termo, e é utilizada para computar o *score máximo possível* que um documento pode atingir de acordo com os termos onde este possa ocorrer.

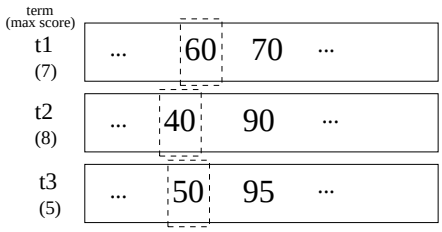

Figura 4.1. Listas invertidas durante o processamento da consulta com os termos $t1$, $t2$ e $t3$. As entradas marcadas estão sendo processadas atualmente. Do lado esquerdo, temos o score máximo de cada lista.

A cada momento no processamento DAD, existe um ponteiro para o próximo documento a ser processado em cada lista invertida associada aos termos da consulta. Por exemplo, se tivermos uma consulta com os termos $t1$, $t2$ e $t3$, sempre existirá um ponteiro para o próximo documento a ser processado em cada uma delas, como ilustrado na Figura 4.1. Nesta figura, os documentos 60, 40 e 50, são os documentos atuais das respectivas listas $t1$, $t2$ e $t3$. O WAND assegura que todas as entradas anteriores de cada lista já foram avaliadas. Assim, como as listas estão organizadas

por *docId*, o menor *docId* (40) ocorre apenas em $t2$, enquanto que o documento 50 ocorre em $t3$ e pode ocorrer em $t2$, e o documento 60 ocorre em $t1$ e pode ocorrer em $t2$ e $t3$.

Sabendo o *score máximo* que um documento pode atingir dentro de cada lista, podemos estivar um *score máximo possível* que cada documento pode atingir em relação à consulta. 40 pode atingir apenas um *score máximo possível* igual ao *score máximo* da lista $t2$, 50 pode atingir a soma do *score máximo* de $t2$ e $t3$, e 60 pode atingir a soma do *score máximo* dos termos $t1$, $t2$ e $t3$. Usando esta informação, podemos descartar os documentos que não sejam capazes de superar o limiar de descarte. Isto é, documentos que não possam atingir um valor de similaridade capaz de colocá-los entre os top-k documentos de resposta já processados até o momento.

Portanto, para descartarmos documentos, estimamos o *score máximo possível* para os documentos apontados por cada lista no momento. O menor *docId* que tenha um *score máximo possível* superior ao *limiar de descarte* é selecionado como candidato a resposta. Este documento é conhecido como pivô. Todas as listas invertidas com o *docId* atual inferior ao pivô selecionado, tem seus ponteiros movidos para a primeira entrada com *docId* maior ou igual ao pivô. Caso uma das listas que tiveram seu ponteiro movido não contenha o pivô, este é descartado, um novo pivô é selecionado e o processo recomeça até que um documento seja avaliado ou que as listas invertidas acabem. Porém, se todas as listas movidas tiverem o *docId* pivô, o documento tem seu *score* completo avaliado e caso seja superior ao *limiar de descarte*, esse é inserido no *heap* de acumuladores. Caso o *heap* já contenha k documentos, o documento de menor *score* é removido para o novo documento ser inserido e o *limiar de descarte* é atualizado com o menor *score* restante no *heap*. Após o pivô ter seu *score* avaliado, todas as listas que o contenham são movidas para o próximo documento e o processo é retomado.

Quando os ponteiros das listas invertidas são movidos, *skiplists* são utilizadas para acelerar o processo, permitindo que blocos de documentos sejam saltados sem a necessidade de descomprimi-los. Por exemplo, suponha que uma lista esteja apontando para o *docId* 60, e o próximo *docId* a ser processado é o 1000. Antes de descomprimir as entradas da lista em busca da próxima entrada maior ou igual ao 1000, primeiramente é utilizado a *skiplist* para procurar o próximo bloco onde o documento 1000 possa ocorrer. Apenas as entradas desse bloco são descomprimidas em busca do documento, diminuindo o custo para percorrer a lista.

Uma característica interessante do WAND é que este pode ser configurado para aplicar uma poda mais agressiva, aplicando-se uma redução no *score máximo* de cada lista, ou aumentando o *limiar de descarte*, de modo que mais documentos sejam descartados. Variando estes parâmetros, faz com que o algoritmo não garanta que o top-k

exato seja retornado. Porém, a velocidade de processamento aumenta.

4.2 Block-Max WAND

Ding & Suel [2011] recentemente revisitaram as idéias apresentadas no WAND, e propuseram um algoritmo de processamento mais eficiente chamado *Block-Max WAND* (BMW). Neste, os documentos de uma lista invertida são agrupados em blocos comprimidos, onde é possível saltar blocos inteiros sem a necessidade de se descomprimir os documentos do bloco. Cada bloco contém o *score máximo possível* dentre os documentos do bloco. Este valor é similar ao valor calculado no WAND para a lista, porém cada bloco tem esse valor pré-calculado considerando apenas os documentos do bloco, e é armazenado juntamente com as entradas da *skiplist*. Sempre que os documentos de um bloco não tenham *score máximo* suficiente para entrar no top-k de resposta, todos os documentos do bloco são descartados, diminuindo os custos de processamento. Podemos observar a organização do índice utilizada pelo BMW na Figura 4.2. Nesta, os documentos da lista são agrupados em blocos de 128 documentos, e para cada bloco temos uma entrada na *skiplist* contendo um ponteiro para descompressão e o *score* máximo registrado entre os documentos do bloco.

Os autores apresentam experimentos indicando que o BMW tem um desempenho no processamento de consultas significativamente superior a outros trabalhos, sendo atualmente o método mais rápido de processamento encontrado na literatura.

O BMW é baseado no algoritmo WAND e utiliza a mesma abordagem para seleção de um documento pivô durante o processamento. A poda de documentos é feita utilizando duas fontes de informação: (i) O *score* máximo da lista, chamado de *score máximo global*, assim como ocorre no WAND; e (ii) O *score* máximo encontrado dentro de cada bloco de documentos, chamado de *score máximo do bloco*.

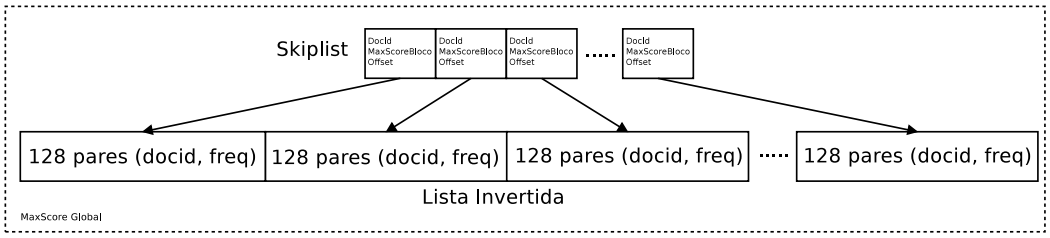

Figura 4.2. Organização da lista invertida utilizada pelo BMW.

A idéia básica do BMW é tirar vantagem da informação (ii) para acelerar o processamento de consultas. Uma vez que um documento é selecionado como pivô, o algoritmo utiliza a informação de *score máximo do bloco* para descartar documentos que

não tenham chance de estar no top-k de resposta. Como cada *score máximo do bloco* é estimado considerando apenas pequenos conjuntos de documentos da lista, o valor tende a ser menor que o *score máximo global* da lista, levando assim a uma estimativa mais precisa do *score máximo possível* de um documento. O algoritmo permite que blocos inteiros sejam descartados baseado no *score* máximo do mesmo, por meio de uma operação chamada *movimento de superfície*. Diferentemente de um movimento regular sobre a lista invertida, o *movimento de superfície* move apenas o ponteiro da *skiplist*, sem descomprimir nenhuma entrada da lista invertida, evitando os custos de descompressão.

O BMW começa pela fase de *pivotação*, onde um documento candidato é selecionado como pivô. Esta fase de *pivotação* é similar à feita no WAND. Utilizando o *score máximo global* de cada lista invertida dos termos da consulta.

Antes de descomprimir as listas em busca do documento pivô, o BMW efetua um *movimento de superfície* para alinhar todas as listas invertidas onde o pivô possa ocorrer, de modo que todas as listas estejam posicionadas sobre o bloco onde possivelmente o pivô esteja. Depois de alinhadas, o algoritmo utiliza a informação de *score máximo do bloco*, para estimar um *score máximo local* para o documento. Caso este *score máximo local* seja inferior ao limiar de descarte, o documento é descartado e uma das listas tem seu ponteiro avançado. Somente se o documento tiver o *score máximo local* superior ao limiar de descarte, o BMW efetua a tarefa de descompressão das listas em busca do documento pivô para calcular seu *score* real.

Assim como no WAND, o BMW mantém um *heap* de acumuladores com os top-k documentos de maior similaridade encontrados. O menor *score* presente no *heap* é utilizado como limiar de descarte e é atualizado dinamicamente conforme novos documentos são inseridos no *heap*.

Podemos observar na Listagem 4.1 o pseudo-código do BMW. Entre as linhas 11 e 44, tem o laço de repetição onde as listas são processadas e o conjunto de documentos de resposta é formado. Na linha 13, selecionamos o documento pivô, utilizando o algoritmo de *pivotação* descrito na Listagem 4.2. Na linha 19, efetuamos o movimento de superfície das listas invertidas. Na linha 21, verificamos o *score máximo local*, descrito na Listagem 4.3, considerando os valores de *score máximo* dos blocos de cada lista. Caso o documento tenha um *score máximo possível* maior que o limiar de descarte e todas as listas selecionadas contenham o documento pivô, esse tem seu valor de similaridade calculado na linha 24. Caso o documento tenha *score* maior que o limiar de descarte θ, ele é inserido no *heap* e o valor de θ é atualizado na linha 30. Na linha 34, todas as listas contendo o pivô são movidas para o próximo $docId$. Caso o documento pivô não tenha *score* suficiente na verificação da linha 21, na linha 40 é selecionado

Listing 4.1. BMW

```
1 BMW (queryTerms[1..q], k)
2   Let H be the minimum heap to keep the top k results
3   Let A be the list of documents
4   Let I be the index
5
6   lists ← I(queryTerms);// Gets inverted lists
7   θ ← 0;
8   //Point to the first docId in each list
9   for each {0 ≤ i < |lists|} do Next(lists[i], 0);
10
11  repeat
12    sortByCurrentPointedDocId(lists);
13    p ← Pivoting(lists, θ);
14    if (p == −1) break; //No more candidates
15    d ← lists[p].curDoc;
16    if(d == MAXDOC) break; //End of the list
17
18    //Move only the skip pointers
19    for each {0 ≤ i ≤ p} do NextShallow(lists[i], d);
20
21    if( CheckBlockMax(θ, p) == TRUE)
22      if(lists[0].curDoc == d)
23        doc.docId ← d;
24        doc.score ← Σ_{i=0}^{p} BM25(lists[i]);
25
26        if (|H| < k) H ← H ∪ doc;
27        else if (H_0.score < doc.score)
28          remove H_0; // the one with smallest score
29          H ← H ∪ doc;
30          θ ← H_0.score; //Update the threshold
31        end if
32
33        //Advance all evaluated lists
34        for each {0 ≤ i ≤ p} do Next(lists[i], d+1);
35      else
36        j ← {x|lists[x].curDoc < d ^ |lists[x]| < |lists[y]|, ∀0 ≤ y < p};
37        Next(lists[j], d);
38      end if
39    else
40      d_next ← GetNewCandidate(lists[j], p);
41      j ← {x| |lists[x]| < |lists[y]|, ∀0 ≤ y ≤ p};
42      Next(lists[j], d_next);
43    end if
44  end repeat
45
46  return A
```

Listing 4.2. Pivotação do BMW

```
1 Pivoting (lists, θ)
2   accum ← 0;
3   for each 0 ≤ i < |lists| do
4     accum ← accum + lists[i].max_score;
5     if (accum >= θ)
6       while(i+1<|lists| AND lists[i+1].curDoc == lists[i].curDoc) do
7         i ← i + 1;
8       end while
9       return i
10    end if
11  end for
12  return −1;
```

um *docId* para ser saltado pela lista de maior *idf*. Como as listas de maior *idf* são menores, os saltos são maiores.

Podemos observar na Listagem 4.4 que o algoritmo salta blocos de documentos inteiros caso estes blocos não tenham nenhum documento que possa estar entre os top-k de resposta. Aliado ao fato de ser feito apenas um movimento de superfície sobre as listas invertidas para alinhar os blocos, o BMW é capaz de descartar

Listing 4.3. CheckBlockMax

```
1 CheckBlockMax (lists, p, θ)
2
3   //Sum the max score of each block, that d can appear
4   max ← Σ_{i=0}^{p} lists[i].getBlockMaxScore();
5
6   if(max > θ) return true
7
8   return false
```

Listing 4.4. GetNewCandidate

```
1 GetNewCandidate (lists, p)
2   mindoc ← MAXDOC
3
4   //Selects the lower docId between the blocks boundaries
5   // of the lists already checked
6   for each {0 ≤ i ≤ p} do
7     if(mindoc > lists[i].getDocBlockBoundary())
8       mindoc ← lists[i].getDocBlockBoundary();
9     end if
10  end for
11
12  //Select the lower docId between the lists not checked
13  for each {p + 1 ≤ i < |lists|} do
14    if(mindoc > lists[i].curDoc)
15      mindoc ← lists[i].curDoc;
16    end if
17  end for
18
19  return mindoc; //Return the smallest docId found
```

blocos de documentos sem a necessidade de descompressão de seu conteúdo. A função $getDocBlockBoundary()$ retorna o maior $docId$ presente no bloco de documentos atual da lista.

Capítulo 5

BMW-CS e BMW-t

Neste capítulo, apresentamos a descrição detalhada dos algoritmos propostos para acelerar o processamento de consultas. Nós os chamamos de BMW-CS (*Block-Max WAND with Candidates Selection*) e BMW-t (*Block-Max WAND with threshold*). Os algoritmos utilizam uma estrutura de índice dividido em duas camadas, onde a primeira camada contém as entradas de maior *score* para cada lista invertida do índice.

A idéia de divisão do índice em camadas é similar à apresentada por Ntoulas & Cho [2007]. O principal objetivo dos algoritmos propostos é utilizar a primeira camada do índice para acelerar o processamento de consultas sobre a segunda camada, onde o índice é maior. O algoritmo *BMW-CS* utiliza a primeira camada para computar o conjunto de candidatos da resposta, e a segunda camada é utilizada apenas para calcular o *score* completo dos candidatos desse conjunto. O *BMW-t* utiliza a primeira camada para selecionar um *limiar de descarte* mais alto para ser aplicado durante o processamento de consultas sobre o índice completo.

Na Seção 5.1 descrevemos as estratégias utilizadas para geração do índice em camadas. Na Seção 5.2 detalhamos o algoritmo *BMW-CS*. E na Seção 5.3 descrevemos o *BMW-t*.

5.1 Organização do Índice em Camadas

O índice invertido é dividido em duas camadas. Onde a primeira camada contém apenas as entradas de maior *score* no índice. A segunda camada pode conter o índice completo ou apenas as entradas que não foram selecionadas para a primeira camada. No caso do *BMW-CS*, a segunda camada é composta apenas pelas entradas que não foram selecionadas para a primeira camada. O *BMW-t* utiliza a segunda camada com o índice completo, uma vez que a primeira camada é utilizada apenas para computar

um valor de limiar para acelerar o processamento do índice completo.

Um índice com *skiplists* é gerado para cada uma das duas camadas visando acelerar o processamento de consultas. Para cada bloco de 128 documentos em cada lista invertida, é gerada uma entrada na *skiplist* correspondente armazenando o $docId$ para descompressão e o maior *score* registrado dentre as entradas do bloco. Essa estrutura é similar à estrutura de *skiplists* utilizada pelo BMW.

Para cada termo da coleção, é armazenado o *score máximo* registrado dentre todas as entradas da lista invertida do termo, e o *score mínimo* registrado na lista do termo apenas na primeira camada. O primeiro valor pode ser interpretado como sendo a maior contribuição que o termo pode oferecer para o *score completo* de um documento. O segundo valor pode ser visto como um limite superior de contribuição que uma entrada da lista invertida do termo que não foi selecionada para a primeira camada pode oferecer para o *score completo* do documento.

Podemos observar na Figura 5.1 a organização em duas camadas de uma lista invertida. Cada camada contém uma *skiplist* associada para acelerar o processamento sobre a lista de documentos. Além das informações de *score máximo* de cada bloco e *score máximo* da lista, é armazenada a informação de *score mínimo* registrado dentre as entradas da primeira camada.

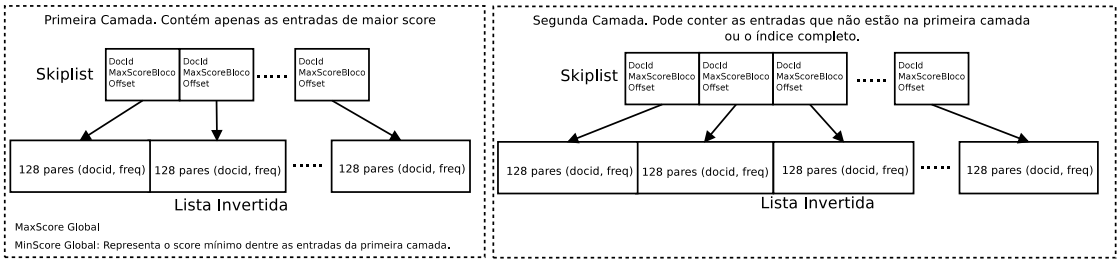

Figura 5.1. Organização da lista invertida dividida em duas camadas.

5.1.1 Gerando o Índice de Candidatos

Neste trabalho, chamaremos a primeira camada do índice de *índice de candidatos*, uma vez que para o algoritmo *BMW-CS*, o conjunto de documentos de resposta é gerado utilizando-se apenas as entradas que estejam nessa camada.

O processo de seleção das entradas para compor o índice de candidatos é um passo chave para o sucesso dos nossos algoritmos. O índice da primeira camada controla o tamanho do conjunto de documentos gerados pela fase de *Seleção de Candidatos*, que detalharemos na Seção 5.2.1. Esta fase tem impacto direto sobre o tempo de processamento de consultas e sobre o conjunto de respostas retornado. Avaliamos duas

estratégias para seleção de entradas para compor o índice de candidatos: 1) Utilizando um *limiar global*; 2) Utilizando um *limiar local*;

Na abordagem global, estima-se um único limiar de corte que seja capaz de dividir o índice completo em duas camadas baseando-se na distribuição de valores de *score* de todas as entradas do índice. A primeira camada é composta apenas pelas entradas que tenham *score* maior que o limiar estimado.

Na abordagem local, estima-se um limiar de corte diferente para cada lista invertida de cada termo da coleção. Apenas as entradas com valor de *score* superior ao limiar de sua respectiva lista invertida são selecionadas para compor o índice de candidatos.

Para cada lista invertida, independente do limiar de corte estimado, foram selecionadas ao menos 1000 entradas para compor o índice de candidatos. Isso foi aplicado para evitar que alguma lista invertida tenha poucas ou nenhuma entrada na primeira camada. O limiar é estimado de acordo com o tamanho do índice de candidatos desejado. Em nossos experimentos, nós variamos o tamanho do índice de candidatos entre 1% e 10% do índice.

Após experimentarmos as duas alternativas, concluímos que os resultados obtidos foram similares, com pequena vantagem para a abordagem global. Portanto decidimos reportar apenas os resultados utilizando esta abordagem para geração do índice de candidatos.

5.2 BMW-CS

A Listagem 5.1 apresenta o pseudo-código geral do nosso primeiro algoritmo, chamado *BMW-CS*. Na primeira fase, o *BMW-CS* utiliza a primeira camada para selecionar documentos que sejam candidatos a estar entre os top-k documentos de resposta a serem retornados. Inicialmente, o conjunto de documentos, denominado $\mathcal{A}$, é gerado pela função *SelectCandidates*, descrita na Listagem 5.2, a qual gera o conjunto de documentos candidatos. Com esse conjunto gerado, é feito uma limpeza removendo os candidatos que não tenham chance de estar dentro do top-k. Essa poda de candidatos diminui sensivelmente o custo de processamento da segunda fase, a qual utiliza um índice consideravelmente maior. Na segunda fase, descrita na Listagem 5.5, é computado o *score completo* dos documentos que foram selecionados na primeira fase.

A principal idéia por trás do *BMW-CS* é tirar vantagem do fato de que a primeira camada contém as entradas de maior *score* dentro de todo o índice. Aproveitando-se dessa característica, o algoritmo busca diminuir os custos de processamento de consul-

Listing 5.1. BMW-CS

```
1  BMW-CS(queryTerms[1..q], k)
2    A ← SelectCandidates(queryTerms, k)
3
4    sort A by score
5    min_score ← A_k.score
6
7    //Remove the candidates with low upper_score
8    for (i = 0 to |A|)
9      if(A_i.upper_score < min_score) remove A_i
10   end for
11
12   R ← CalculateCompleteScore(A, queryTerms, k)
13
14   return R
```

tas reduzindo o montante de dados descomprimidos e avaliados na segunda fase, onde o índice é consideravelmente maior e mais custoso para se processar completamente. Em nossos experimentos, nós mostramos que o algoritmo apresenta um desempenho de processamento bastante superior aos métodos comparados.

Como o algoritmo *BMW-CS* utiliza apenas a primeira camada para gerar o conjunto de documentos que irão compor os top-k documentos de resposta, não existe a garantia de que o top-k exato será retornado. Isto significa que o método é aproximado e não garante que o conjunto de respostas será idêntico ao obtido por um método como o BMW, que avalia os documentos do índice completo para montar o conjunto de respostas. Nos experimentos que serão apresentados no capítulo seguinte, mostramos que o *BMW-CS*, apesar de não garantir o conjunto exato de respostas, gera uma variação no conjunto de resposta praticamente imperceptível para o usuário.

5.2.1 Seleção dos Candidatos

O algoritmo *BMW-CS* seleciona os documentos candidatos à resposta utilizando apenas o índice da primeira camada, ou índice de candidatos. Porém, como as listas invertidas dos termos da coleção na primeira camada não estão completas, um documento que ocorreu apenas na lista invertida de um dos termos da consulta no índice de candidatos pode ocorrer nas demais listas quando se avalia a segunda camada do índice. Portanto, durante a fase de seleção dos candidatos, utilizar um algoritmo ingênuo de processamento como *BMW*, que aplica poda dinâmica diretamente sobre o índice invertido poderia descartar documentos com alto valor de *score*, pois os valores de *score* obtidos processando-se apenas a primeira camada nem sempre são completos.

Para evitar que documentos com potencial para compor o top-k sejam descartados durante a seleção de candidatos, nós modificamos o algoritmo *BMW* para que este considere a possibilidade de que um par $(termo, docId)$ inexistente na primeira camada, possa ocorrer na segunda. Durante a fase de *pivotação* e verificação do *score máximo*

possível de um candidato, para cada par inexistente, é somado o valor de *score mínimo* armazenado junto ao termo descrito anteriormente.

Este valor representa a contribuição máxima que uma entrada não encontrada na primeira camada teria caso fosse encontrada na segunda camada. Assim, para cada documento candidato, nós temos o valor de *score* obtido na primeira camada e o valor de *score máximo* que o documento pode ter ao se utilizar o índice completo. Utiliza-se uma estrutura de *heap* para armazenar os top-k documentos de maior *score* durante o processo.

O menor score do *heap* é utilizado como *limiar de descarte* para que se possa podar dinamicamente entradas sem *score máximo* suficiente para fazer parte do top-k. Todos os documentos com *score máximo* superior ao *limiar de descarte* são adicionados ao conjunto de candidatos. É importante ressaltar que apesar do método utilizar o *score* dos documentos obtidos na primeira camada para definir o *limiar de descarte*, os documentos são podados apenas se tiverem seu *score máximo* inferior a este limiar.

Podemos observar o pseudo-código do algoritmo de seleção de candidatos na Listagem 5.2. O algoritmo começa selecionando as listas invertidas a serem processadas (linha 6). O *limiar de descarte*, Θ, é inicializado com zero (linha 7) e é atualizado conforme o conjunto do *heap* muda quando está cheio (linha 32). Na linha 9, cada ponteiro das listas invertidas é inicializado com o primeiro documento da lista. A função $Next(l, d)$ busca por meio da *skiplist* associada à lista l, o bloco onde o documento d possa ocorrer. Depois descomprime o bloco em busca da primeira entrada com $docId >= d$ e armazena a entrada encontrada em $l.curDoc$.

As listas manipuladas na Listagem 5.2 são representadas pelo vetor $lists$. Na linha 12 esse vetor é ordenado em ordem crescente de $curDoc$. Na linha 13, computamos o próximo documento que tem chances de estar presente no top-k de resposta. Para isso efetuamos a *pivotação*, que é um passo central do algoritmo BMW. Porém na *pivotação* do BMW-CS, é considerada a possibilidade de que uma entrada que não ocorreu na primeira camada possa ocorrer na segunda. Para tanto, computamos o *score máximo*, descrito anteriormente, do documento considerando essa possibilidade. O cômputo do score máximo é descrito na Listagem 5.3. Nas linhas 15 à 17, nós testamos essa condição e selecionamos o documento que será pivô.

A linha 19 utiliza a função $NextShallow$ para prover os ponteiros de cada lista invertida. Esta função é a mesma proposta por Ding & Suel [2011] para o BMW, descrita anteriormente na Seção 4.2. Diferentemente da função $Next$, que busca o bloco onde o documento possa ocorrer e descomprime suas entradas atrás do documento, a função $NextShallow$ apenas posiciona o ponteiro da lista sobre o bloco onde possivelmente determinado documento possa estar, sem fazer nenhuma tarefa de descompres-

Listing 5.2. Seleção dos Candidatos

```
1  SelectCandidates (queryTerms[1..q], k)
2    Let H be the minimum heap to keep the top k results
3    Let A be the list of candidates
4    Let I_cand be the first tier index
5
6    lists ← I_cand(queryTerms);// Gets inverted lists
7    θ ← 0;
8    //Point to the first docId in each list
9    for each {0 ≤ i < |lists|} do Next(lists[i], 0);
10
11   repeat
12     sortByCurrentPointedDocId(lists);
13     p ← Pivoting(lists, θ);
14     if (p == −1) break; //No more candidates
15     d ← lists[p].curDoc;
16     if(d == MAXDOC) break; //End of the list
17
18     //Move only the skip pointers
19     for each {0 ≤ i ≤ p} do NextShallow(lists[i], d);
20
21     if( CheckBlockMax(θ, p) == TRUE)
22       if(lists[0].curDoc == d)
23         doc.docId ← d;
24         doc.score ← Σ_{i=0}^{p} BM25(lists[i]);
25         doc.upper_score ← doc.score +
26                           Σ_{i=p+1}^{|lists|} lists[i].min_score;
27
28         if (|H| < k) H ← H ∪ doc;
29         else if (H_0.score < doc.score)
30           remove H_0; // the one with smallest score
31           H ← H ∪ doc;
32           θ ← H_0.score; //Update the threshold
33         end if
34
35         //Insert only documents with possible score > θ
36         if (θ <= doc.upper_score)
37              doc.terms ← queryTerms[0..p];
38              A ← A ∪ doc;
39              if ({∃dLow ∈ A| dLow.upper_score < θ})
40                   A ← A − dLow;
41              endif
42         endif
43         //Advance all evaluated lists
44         for each {0 ≤ i ≤ p} do Next(lists[i], d+1);
45       else
46         j ← {x|lists[x].curDoc < d ^
47              |lists[x]| < |lists[y]|, ∀0 ≤ y < p};
48         Next(lists[j], d);
49       end if
50     else
51       d_next ← GetNewCandidate(lists[j], p);
52       j ← {x| |lists[x]| < |lists[y]|, ∀0 ≤ y ≤ p};
53       Next(lists[j], d_next);
54     end if
55   end repeat
56
57   return A
```

Listing 5.3. Pivotação do BMW-CS

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

Listing 5.4. CheckBlockMax do BMW-CS

```
1 CheckBlockMax (lists, p, θ)
2
3   //Sum the max score of each block, that d can appear
4   max ← Σ_{i=0}^{p} lists[i].getBlockMaxScore();
5
6   //Add the min score of the lists that d may appear in the full index
7   max ← max + Σ_{i=p+1}^{|lists|} lists[i].min_score;
8   if(max > θ) return true
9   return false
```

são. Utilizando esta função, podemos descartar blocos inteiros sem a necessidade de descomprimir nenhuma entrada. Na linha 21, chamamos a função $CheckBlockMax$, detalhada na Listagem 5.4, que também foi modificada para lidar com o fato da lista invertida estar incompleta nesta fase de processamento.

O restante do algoritmo verifica quando um documento contém *score* suficiente para ser incluído no conjunto top-k de respostas. Na linha 25, somamos o *score mínimo* das listas onde o documento não foi encontrado na primeira camada. Com isso temos o *score máximo* possível para o documento. Entre as linhas 28 e 33, nós atualizamos o *limiar de descarte* por meio do *score* do documento avaliado. O *score máximo* é utilizado na linha 26 para verificar se o documento deveria ser inserido no conjunto de candidatos $\mathcal{A}$.

O *limiar de descarte* θ varia conforme os documentos são processados. Portanto, quando adicionamos um documento ao conjunto $\mathcal{A}$, nós verificamos também se existe algum documento em $\mathcal{A}$ com *score máximo* inferior ao valor atual de θ. Este procedimento evita um gasto desnecessário de memória com acumuladores armazenando documentos em $\mathcal{A}$ que futuramente serão descartados. A função $GetNewCandidate$ é a mesma utilizada pelo BMW, descrita na Listagem 4.4.

Ao final do algoritmo de seleção de candidatos, a lista de documentos candidatos a resposta $\mathcal{A}$ é retornada. O *score* completo dos documentos desta lista são então computados utilizando-se a segunda camada, que contém o restante das entradas do índice.

5.2.2 Computando o Ranking Final

Na função $CalculateCompleteScore$ (Listagem 5.5), o *score completo* dos candidatos com entradas não encontradas na primeira camada, é computado utilizando-se o índice da segunda camada, o qual contém as entradas não armazenadas na primeira camada. Para evitar custos desnecessários com descompressão, nós utilizamos o movimento de superfície $NextShallow$ para alinhar as listas sobre o $docId$ de cada documento a ser

Listing 5.5. Calculo do Score Completo

```
1  CalculateCompleteScore( A, queryTerms[1..q], k)
2    Let H be the minimum heap to hold the k most relevant candidates
3    Let I_second_tier be the second index
4    lists ← I_second_tier(queryTerms) //Select the terms lists
5    θ ← 0
6
7    H ← {}
8
9    sort A by docId;
10
11   for each {0 ≤ i < |A|} do
12     if (A_i.score < A_i.upper_score)
13      local_upper_score ← A_i.score;
14
15      for each {0 ≤ j < |lists|} do
16       if (queryTerm[j] ∉ A_i.terms)
17        NextShallow(lists[j], A_i.docId);
18        local_upper_score ← local_upper_score +
19                              lists[j].getBlockMaxScore();
20       end if
21      end for
22
23      if (local_upper_score > θ)
24       for each {0 ≤ j < |lists|} do
25        if (queryTerm[j] ∉ A_i.terms)
26         Next(lists[j], A_i.docId);
27        end if
28       end for
29       //Complete the score with the missing lists
30       for each {0 ≤ x < |lists||lists[x].curDoc = A_i.docId} do
31         A_i.score ← A_i.score + BM25(lists[x]);
32       end for
33      end if
34     end if
35
36     if (θ < A_i.score)
37       if (|H| < k)
38         H ← H∪ A_i;
39       elseif (H_0.score < A_i.score)
40         remove H_0;
41         H ← H∪ A_i;
42         θ ← H_0.score;
43       end if
44     end if
45   end for
46
47   sort H by score;
48   return H;
49 end
```

avaliado. Então, uma segunda checagem do *score máximo possível* é feita para verificar se o documento pode fazer parte do top-k resultados de resposta. Um documento só é avaliado quando o seu *score máximo possível* supera o atual *limiar de descarte* do processo. Assim como na primeira fase, nós mantemos um *heap* mínimo com os documentos de maior score registrados até o momento, e o menor *score* dentre os documentos armazenados no *heap* é utilizado como o *limiar de descarte* θ para podar candidatos sem *score* suficiente.

Como o conjunto de candidatos é pequeno, devido à poda de acumuladores feita nas fases anteriores, e apenas documentos com *score* incompleto serem avaliados nesta etapa, o custo de processamento é baixo, tornado esta fase extremamente rápida mesmo se processando um índice consideravelmente maior que o índice da primeira camada.

5.3 BMW-t

O segundo algoritmo proposto, *BMW-t*, utiliza a primeira camada apenas para determinar um limiar de descarte θ inicial utilizado pelos métodos WAND e BMW. Nestes métodos, o *limiar de descarte* é inicializado com zero e vai crescendo conforme documentos de maior *score* são avaliados. Como consequência, o processamento de consultas descarta poucas entradas no início do processo, uma vez que o limiar é baixo. Nós então propusemos a utilização de uma primeira camada de índice apenas para estimar um *limiar de descarte* alto. Esse limiar é posteriormente utilizado desde o início do processamento de consultas sobre o índice completo. Com o limiar alto desde o início do processo, mais documentos serão descartados, aumentando-se a velocidade de processamento do método, mesmo tendo uma etapa a mais para computar este limiar inicial.

Neste cenário, a primeira camada é uma pequena porcentagem do índice completo. Enquanto que a segunda camada contém o índice completo, e não apenas as entradas restantes como ocorre no algoritmo BMW-CS. O algoritmo original do BMW Ding & Suel [2011] é utilizado nas duas etapas de processamento do BMW-*t*.

Diferentemente do algoritmo *BMW-CS*, que retorna um conjunto de respostas aproximado, o *BMW-t* preserva o conjunto exato de respostas pois utiliza a primeira camada apenas para computar um *limiar de descarte*, e o conjunto de respostas é computado utilizando-se o índice completo.

5.4 Complexidade

Analisando o algoritmo BMW apresentado na Seção 4.2, temos que no pior caso todas as entradas das listas invertidas de uma consulta serão avaliadas. Sendo assim podemos definir como n o número total de entradas que serão descomprimidas. Cada entrada descomprimida é avaliada e pode ou não ser inserida no conjunto de acumuladores de tamanho k, onde k representa o número de respostas solicitadas. Utilizando uma estrutura de *heap*, podemos efetuar inserções no conjunto de acumuladores em tempo $\log k$. Como no pior caso todas as listas de uma consulta contém entradas distintas e todas as entradas passam pelo limiar de corte, temos que o custo de processamento de uma consulta utilizando o BMW é de $O(n \log k)$.

O algoritmo BMW-*t* tem o mesmo custo de processamento do BMW original, sendo que existe uma fase anterior onde o BMW-*t* executa o BMW sobre um índice consideravelmente menor para se obter um limiar de descarte inicial para o BMW ser executado sobre o índice completo. Como o tamanho do índice utilizado para se

estimar este limiar inicial é de apenas 1% do índice completo, o custo final do BMW-t é também $O(n \log k)$.

O algoritmo BMW-CS tem uma análise um pouco diferente. Para a fase de seleção de candidatos temos n_1 sendo o número de entradas das listas invertidas da primeira camada. Para cada entrada avaliada, temos o custo $logk$ para inseri-la no *heap* de acumuladores $\mathcal{H}$, e o custo $logA$ para inserir na lista de candidatos $\mathcal{A}$, onde $A > k$. Sendo assim, o custo da fase de seleção de candidatos é de $O(n_1 logA)$. Na prática A se aproxima do valor de k.

Após a primeira fase, existe uma poda de acumuladores, onde a lista de candidatos $\mathcal{A}$ é primeiramente ordenada com um custo $O(AlogA)$, e depois tem os candidatos de *score* insuficiente removidos com custo $O(A)$. Assim temos que a poda de acumuladores tem custo $O(AlogA)$.

A fase de cálculo do *score* completo do conjunto de candidatos é efetuada sobre o índice da segunda camada. Neste índice temos que o número total de entradas a serem avaliadas é n_2. Primeiramente o algoritmo ordena a lista de candidatos $\mathcal{A}$ pelo $docId$ novamente com custo $O(AlogA)$. No pior caso, todas as entradas serão descomprimidas em busca dos documentos do conjunto $\mathcal{A}$. Porém, apenas as entradas do conjunto A serão avaliadas. Assim, podemos observar que temos custo $O(n_2 logk)$ para descomprimir todas entradas e inseri-las no *heap* de acumuladores $\mathcal{H}$. Portanto, o custo da fase de cálculo do *score* completo é de $O(n_2 logk + AlogA)$.

O custo final do BMW-CS é de $O(n_1 logk + n_2 logk + AlogA)$. Como $n_2 = cn$ e $n_1 = (1-c)n$, com a constante $c < 1$, e como A podemos reduzir o custo para $O(nlogk)$, onde n é o número de entradas da segunda camada. É importante observar que todos os algoritmos possuem funções auxiliares para ordenação da lista de termos, cálculo do valor de similaridade e verificação do *score* máximo de um documento. Como o custo destas funções é relativo ao número de termos da consulta, valor usualmente pequeno e que não costuma variar, podemos considerá-los constantes.

Outro fator importante a se ressaltar é que na prática os métodos têm um desempenho muito superior do pior caso devido ao uso dos movimentos de superfície, feito com o uso das *skiplists*, que diminuem drasticamente o número de entradas avaliadas durante o processamento.

Capítulo 6

Experimentos

Neste capítulo apresentamos os experimentos para avaliação do desempenho dos métodos de processamento de consultas apresentados no capítulo anterior. Na Seção 6.4 discutimos os resultados atingidos nos diversos cenários avaliados.

6.1 Coleção

Para avaliarmos o desempenho dos algoritmos implementados foi utilizada a coleção de referência TREC GOV2, composta por aproximadamente 25 milhões de páginas coletadas do domínio .gov no início de 2004. A TREC GOV2 possui 426 GB de texto, compostos por páginas HTML e texto extraído de páginas no formato PDF e *postscript* (PS). O índice invertido gerado possui aproximadamente 7 GB de listas invertidas e o vocabulário é composto por cerca de 4 milhões de termos distintos. Nós aplicamos o algoritmo para redução de sufixos *Porter Steammer* Porter [2006], para reduzir o tamanho do vocabulário. O índice gerado é comprimido e armazena, em cada entrada, um par $(docId, freq)$. Onde o $docId$ armazena a diferença do $docId$ atual para o anterior, diminuindo o valor do inteiro a ser comprimido.

Selecionamos aleatoriamente um conjunto de 1000 consultas do conjunto *TREC 2006 efficiency queries*. Durante o processamento de consultas, o índice inteiro é carregado para a memória. Isso evita qualquer interferência na avaliação do tempo de processamento de uma consulta. Todas as configurações foram escolhidas por serem similares às adotadas em trabalhos similares [Strohman & Croft, 2007; Ding & Suel, 2011], o que torna mais fácil a comparação entre os métodos estudados.

6.2 Parâmetros

Escolhemos utilizar o modelo Okapi BM25, descrito na Seção 2.2, como função de similaridade. Porém, é importante ressaltar que nosso método poderia ser aplicado com outros modelos Salton et al. [1974]. As *skiplists* geradas possuem uma entrada para cada bloco de 128 documentos. Em cada entrada, é armazenado o $docId$ do primeiro documento do bloco, para ajudar na descompressão, e o *score máximo* registrado entre os documentos do bloco. Nós experimentamos *skiplists* com blocos de 64 documentos, e os resultados foram similares, portanto decidimos reportar apenas os resultados utilizando *skiplists* para blocos de 128 documentos.

Para a geração da primeira camada (índice de candidatos), nós variamos o tamanho $\Delta\%$ entre 1 e 20 porcento do índice completo nos experimentos. A variação do Δ é útil no estudo do seu custo e benefício por impactar diversos parâmetros que aferam o tempo final de processamento, tais como o número de acumuladores necessários para processar consultas e o tempo gasto no processamento das duas camadas de índice.

Outro parâmetro avaliado foi o número k de respostas retornadas pelo sistema. Nós avaliamos os algoritmos retornando o top-10 e o top-1000. A recuperação dos top-1000 foi incluída para simular um ambiente onde o conjunto top-1000 é utilizado como entrada para um método de ordenação mais sofisticado, utilizando aprendizado de máquina por exemplo. O top-10 foi incluído para simularmos um cenário mais comum, onde o usuário está interessado apenas em uma pequena lista de resultados para a sua consulta, e o sistema utiliza apenas uma função de similaridade para ordenar os documentos de resposta.

Os algoritmos foram avaliados em termos de tempo de resposta, quantidade de acumuladores utilizados para o processamento de consultas, quantidade de entradas decodificadas, e o MRRD do método, descrito na Equação 6.1.

O MRRD é a mesma medida utilizada por Broder et al. [2003] para avaliar a distância entre dois conjuntos de resposta retornados por um método que preserva o top-k de resposta comparado com um método que não preserva. Por meio do valor de MRRD, podemos mensurar a variação no conjunto de resposta retornado pelo algoritmo aproximado implementado no trabalho. Um conjunto idêntico tem valor de MRRD igual a zero, enquanto dois conjuntos totalmente diferentes possuem valor de MRRD igual a 1.

$$MRRD(B,P) = \frac{\sum_{i=1,d_i \in B-P}^{k} 1/i}{\sum_{i=1}^{k} 1/i} \tag{6.1}$$

6.3 Baselines

O principal *baseline* implementado neste trabalho foi o BMW, proposto por Ding & Suel [2011]. Como os nossos métodos não preservam o conjunto de resposta exato retornado, nós também incluímos como *baseline* uma versão do WAND, chamada WAND-f que computa um conjunto top-k aproximado. Esta versão, descrita em Broder et al. [2003], consiste em aumentar artificialmente o limiar de descarte do método de acordo com um fator f, fazendo com que o método descarte mais candidatos e melhores o tempo de processamento. No entanto, os resultados obtidos com essa variação do WAND aproximado foram inferiores aos resultados do BMW. Nós então aplicamos a mesma idéia para o método BMW tornando este um método aproximado também. Nós denominamos o método de BMW-f.

Como o BMW é baseado no WAND, nós aumentamos artificialmente o *limiar de descarte* do método assim como foi feito no WAND. Com um limiar de descarte mais alto, mais entradas serão descartadas pelo BMW-f, diminuindo o custo de processamento. Quando f é 1, o conjunto exato de respostas é preservado. Para qualquer valor de f maior que 1, o conjunto retornado torna-se aproximado.

Uma dúvida que poderia ser levantada seria se o conjunto top-k de resposta retornado por um método aproximado poderia ser tão bom quanto o conjunto retornado, processando apenas a primeira camada do índice, e descartando a segunda camada do processamento. Nós incluímos como *baseline*, o BMW utilizando apenas a primeira camada do índice para processamento. Nós chamamos o método de BMW-SP. Porém, para atingir resultados competitivos, a porção $\Delta\%$ do índice selecionado para a primeira camada foi maior que a utilizada pelos demais métodos. Uma vez que porções pequenas do índice causavam grande variação no conjunto de respostas comparado ao conjunto de resposta exato.

6.4 Resultados

Nós começamos a reportar o resultado dos nossos experimentos respondendo a possíveis dúvidas sobre a vantagem de se utilizar o algoritmo BMW-CS, o qual não garante que o top-k seja preservado, quando comparado com os demais métodos que geram um top-k aproximado. A Figura 6.1 apresenta os resultados de MRRD quando variamos o parâmetro k, que determina o número de resultados retornados. Nós apresentamos as variações do BMW-CS utilizando 1%, 5% e 10% do índice como a primeira camada para formar o índice de candidatos. A porcentagem do índice utilizado pelo método BMW-SP varia entre 10%, 30% e 50% do índice completo. Apesar da variação de MRRD

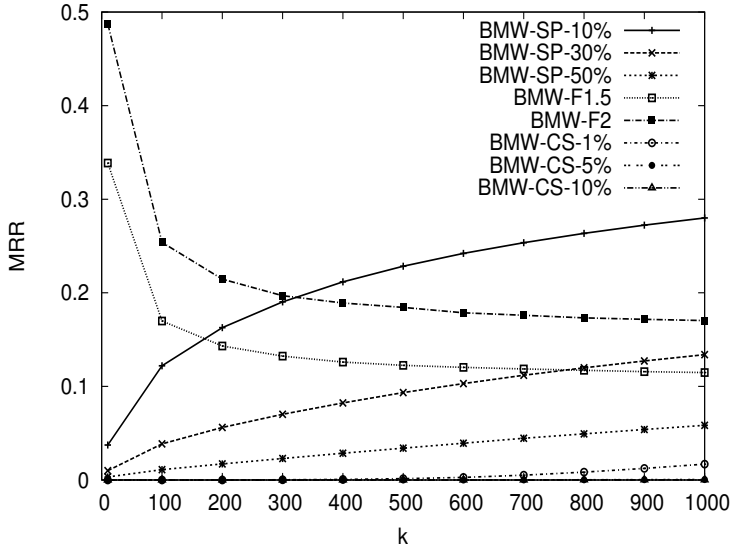

Figura 6.1. Valores de MRRD dos algoritmos comparados com o rank exato

nos métodos de *baseline* ser significativamente maior que nos nossos métodos, nós reportamos o desempenho nos experimentos de eficiência para o BMW-SP utilizando 50% do índice, e BMW-f utilizando f=1.5.

Na Figura 6.1 também apresentamos os valores de MRRD para o método BMW-f, variando o parâmetro f entre 1.5 e 2.0. Como podemos observar, os valores de MRRD para o BMW-f são piores que os obtidos pelo método BMW-CS. Um fator f menor resulta em valores de MRRD melhores. Porém, o custo de processamento é maior. Como o fator f é utilizado para incrementar o limiar de descarte artificialmente, podemos observar que os valores de MRRD diminuem conforme o tamanho do conjunto top-k de resposta aumenta, diferentemente dos demais métodos. Isso se deve ao fato de que o limiar de descarte é consideravelmente maior quando se processa apenas o top-10. Portanto, aumentando este limiar resulta no descarte de muitos documentos de *score* alto. Enquanto que quando se processa o top-1000, o limiar de descarte é menor e assim menos documentos de *score* alto são descartados.

É importante ressaltar que os algoritmos BMW-f e BMW-SP não foram explicitamente propostos na literatura, e foram incluídos nos experimentos apenas para evitar dúvidas sobre a possibilidade de variação destes *baselines* para processar um top-k aproximado. Métodos de poda estática mais robustos já foram propostos em diversos trabalhos conforme discutimos na Seção 3.2.

A escolha da porcentagem de entradas que serão incluída na primeira camada do índice utilizado pelo BMW-CS tem impacto em fatores principais do processamento: 1) tempo de resposta; 2) valor de MRRD; 3) número médio de acumuladores neces-

sários para se processar uma consulta. A variação desses parâmetros pode ser vista na Figura 6.2. Como podemos observar, os valores de MRRD, quando computamos o top-1000, diminuem conforme o tamanho da primeira camada aumenta. Enquanto que processando o top-10, o valor de MRRD foi zero em todos os tamanhos avaliados. O tempo tende a crescer conforme o tamanho da primeira camada aumenta. Porém, o número de acumuladores apresenta um comportamento mais complexo. Para o top-1000, o número de acumuladores inicialmente cresce conforme o tamanho da primeira camada aumenta. Então em certo ponto, esse começa a diminuir, a partir do momento que a fase de seleção de candidatos passa a computar o *score completo* de mais candidatos e assim efetuar uma poda mais precisa, reduzindo o tamanho do conjunto de candidatos retornados para a segunda fase.

A Figura 6.2 (c) e (d) apresenta a quantidade de memória gasta com acumuladores pelo BMW-CS. Podemos ver que o número médio de acumuladores é limitado em algumas vezes o tamanho do top-k requisitado.

Olhando para os resultados gerais apresentados na Figura 6.2, podemos concluir que o melhor tamanho para a primeira camada no BMW-CS é 2% quando otimizamos o método para computar apenas o top-10, e 10% quando queremos computar o top-1000. Esses parâmetros nos dão a melhor combinação com um baixo número de acumuladores necessários, baixo tempo de processamento e baixo MRRD. Ainda assim, podemos observar que mesmo utilizando outros tamanhos para a primeira camada, nosso método continua consideravelmente mais rápido que os *baselines* comparados.

Finalmente, um comentário sobre os valores de MRRD obtidos pelo BMW-CS é que a diferença no top-k retornado é praticamente imperceptível para um usuário, mesmo considerando altos valores de k. Para valores de $k = 1000$, o erro continua inferior a 0.0001 em termos de MRRD para um índice de candidatos com 10% das entradas do índice completo. Para melhor ilustrar o que esse erro significa, analisando os resultados em detalhes, observamos que 90% das consultas processadas não apresenta variações no top-1000 retornado. Enquanto que no top-10 e top-100, para qualquer tamanho da primeira camada avaliado, não ocorreu modificações no conjunto de resposta, tendo o valores de MRRD permanecido em zero. Somente a partir do top-200, que teve 99.9% das respostas preservadas, o método começou a apresentar variação no conjunto retornado.

Como o algoritmo BMW-t preserva o conjunto exato de resposta, não foi necessário fazer um estudo sobre os valores de MRRD. Para esse algoritmo avaliamos apenas o desempenho em termos de tempo de processamento. O melhor tempo de processamento atingido foi utilizando apenas 1% do índice completo para selecionar o limiar de descarte inicial. O tempo de processamento cresce conforme o tamanho da primeira

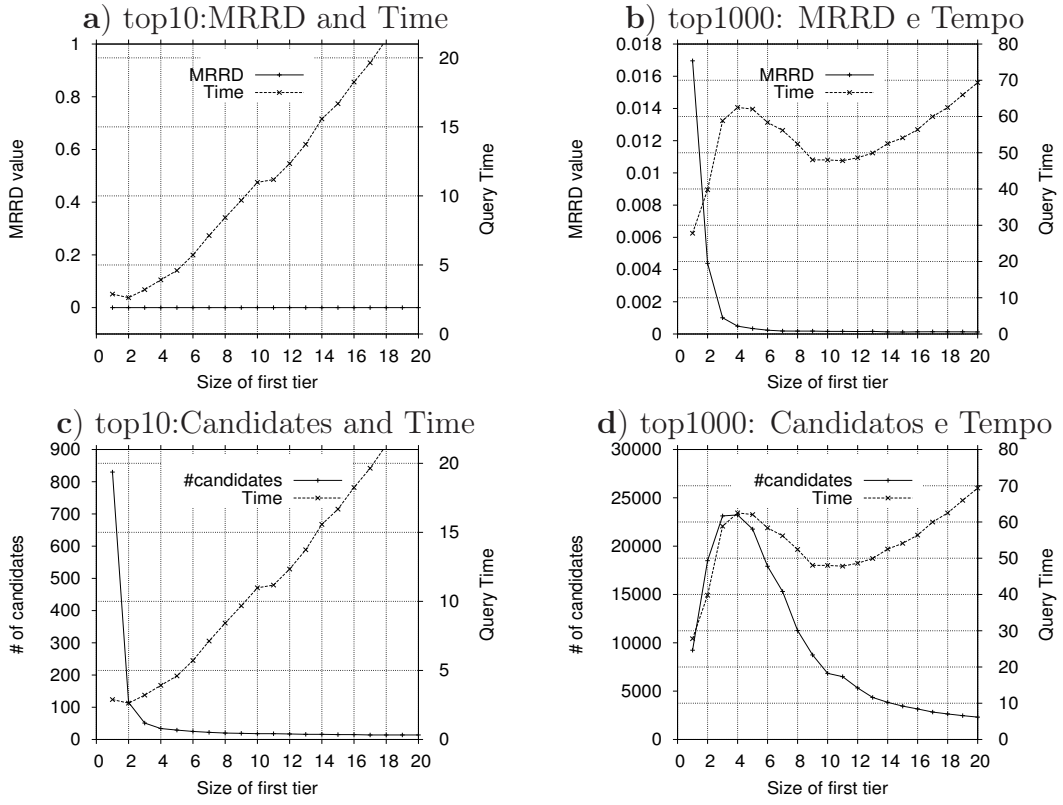

Figura 6.2. Variação no tempo em MRRD (a, b); e entre o tempo e o número de acumuladores (c, d) quando se processa a primeira camada para o método BMW-CS usando tamanhos da primeira camada distintos.

camada aumenta. É importante ressaltar que no caso do BMW-t, a primeira camada é utilizada apenas para se selecionar um limiar de descarte inicial, enquanto que a segunda camada contém o índice completo. O algoritmo BMW original é utilizado nas duas fases de processamento.

Na Tabela 6.1 podemos observar o desempenho dos algoritmos em termos de MRRD, inteiros decodificados e tempo quando processamos o top-10 e o top-1000. A Tabela 6.1 mostra que o BMW-CS apresenta valores insignificantes de MRRD. Conforme já discutido, isso significa que o conjunto de resposta retornado pelo algoritmo é praticamente o mesmo retornado por um método exato. Enquanto que a velocidade de processamento do método é consideravelmente superior aos *baselines* BMW e BMW-f.

Podemos verificar que o BMW-t apresenta uma melhoria de 10% no tempo de processamento de uma consulta, quando comparado com o BMW original. A versão aproximada BMW-f, utilizando um fator f=1.5, processa consultas 20% mais rápido que o BMW exato, porém seus valores de MRRD são elevados.

Observando a Tabela 6.1, podemos ver que o BMW-CS mesmo sendo um método

aproximado, foi capaz de preservar o conjunto de respostas no top-10. Em termos de velocidade, o BMW-CS chega a ser 40 vezes mais rápido que o BMW quando computando o top-10, utilizando 2% do índice na primeira camada. E aproximadamente 4.75 vezes mais rápido quando computando o top-1000, utilizando 10% do índice para a primeira camada. Estes ganhos também podem ser observados quando analisamos o número de entradas decodificadas, que é uma das tarefas mais custosas no processamento de consultas quando o índice se encontra em memória principal.

	top 10			top 1000		
Algoritmo	MRRD	Decodificados	Tempo	MRRD	Decodificados	Tempo
BMW	0	2353066	100.1	0	5032834	226.0
BMW-f1.5	0.3386	1797451	78.5	0.1149	4463099	193.7
BMW-SP-50	0.0032	1700488	79.8	0.0584	3495885	174.8
BMW-t	0	2082782	89.5	0	4799477	205.7
BMW-CS-2	0	48989	2.4	0.0043	1258859	39.8
BMW-CS-10	0	217627	10.4	0.0001	921687	47.6

Tabela 6.1. MRRD, número de entradas decodificadas, e tempo(ms) atingidos pelos baselines (BMW, BMW-f, BMW-SP) e pelos algoritmos propostos (BMW-t, BMW-CS), quando computando o top-10 e o top-1000.

O principal impacto na velocidade de processamento, quando aumentamos o tamanho k do conjunto a ser retornado, é que os limiares de poda se tornam menores pois mais documentos podem passar a ser candidatos a resposta, uma vez que o conjunto requisitado é maior. Nesse cenário, observamos que nossos algoritmos superam os *baselines*, e continuam apresentando valores baixos de MRRD. O algoritmo BMW-f aproveita o fato do limiar de poda ser menor para computar um top-k de resposta mais próximo do conjunto retornado por um método exato, como o BMW.

A Tabela 6.2 apresenta o desempenho dos algoritmos para diferentes tamanhos de consulta, computando o top-10 e o top-1000. Novamente o BMW-CS foi o mais rápido em todos os casos. Porém os ganhos foram menores para consultas com mais de 5 termos.

Uma explicação para o ganho ser menor para consultas longas no BMW-CS, é que na primeira fase do processamento, onde é gerado o conjunto de candidatos, o *score máximo* de cada documento candidato passa a ser maior, uma vez que sempre somamos o *score mínimo* de cada entrada que esteja faltando na primeira camada. Isso faz com que muitos documentos passem a ser candidatos a resposta. Como o índice da primeira camada é consideravelmente menor que o índice da segunda camada, o número de entradas não encontradas na primeira camada é maior para consultas longas. Sendo assim, o *score máximo* fica muito alto, diminuindo a capacidade de poda do algoritmo.

Algoritmo	Tempo(ms)				
	2	3	4	5	>5
	top 10				
BMW	12.0	46.6	100.5	197.9	442.7
BMW-f1.5	7.8	34.0	77.3	155.5	367.7
BMW-SP-50	10.3	38.7	82.4	160.1	331.6
BMW-t	10.4	41.1	87.7	179.3	401.7
BMW-CS-2	0.58	1.42	2.42	3.74	9.96
BMW-CS-10	2.3	5.7	11.6	19.7	36.8
	top 1000				
BMW	37.9	122.3	243.6	437.9	848.6
BMW-f1.5	29.9	102.5	206.3	390.1	725.0
BMW-SP-50	32.6	102.1	192.0	333.5	610.3
BMW-t	29.8	104.6	220.9	406.5	803.5
BMW-CS-2	8.49	20.62	44.58	79.07	139.1
BMW-CS-10	14.6	29.3	51.8	82.2	160.8

Tabela 6.2. Tempo de processamento para consultas de tamanhos distintos.

Capítulo 7

Conclusões e Trabalhos Futuros

Os algoritmos propostos e estudados neste trabalho superam em desempenho os atuais algoritmos *estado-da-arte* para processamento de consultas DAD. BMW-CS apresenta a vantagem de ser 40 vezes mais rápido que o BMW para computar o top-10, e 4.75 vezes mais rápido para computar o top-1000. Apesar do método não garantir que o conjunto de resposta seja preservado, observamos através de experimentos que a variação no conjunto de resposta é imperceptível para o usuário, uma vez que os valores de MRR são muito pequenos. Portanto em situações onde preservar o top-k não seja obrigatório ou essencial, a utilização do BMW-CS se torna uma alternativa bastante vantajosa do ponto de vista de velocidade de processamento.

O preço pago por um processamento mais eficiente é o maior consumo de memória para manter o conjunto de acumuladores durante o processamento da consulta. Como mostramos nos experimentos, o número de candidatos que o algoritmo gera durante o processamento, em alguns casos sendo cerca de 10 vezes maior, não chega a ser um fator proibitivo para a utilização do algoritmo. Em sistemas de busca, geralmente uma máquina processa mais de uma consulta ao mesmo tempo, e a redução no tempo de processamento de uma consulta acaba por compensar o custo de memória extra uma vez que a taxa de consultas atendidas em um espaço de tempo tende a aumentar.

O segundo algoritmo proposto, BMW-t, apresenta a vantagem de preservar o conjunto de resposta retornado. Contudo, com um ganho pequeno, em relação ao método BMW original, sendo 10% mais rápido.

É importante ressaltar que em um ambiente mais realístico, onde o índice completo não possa ser mantido em memória, nossos métodos de processamento atingiriam ganhos ainda maiores, uma vez que a primeira camada, por ser pequena, pode ser sempre mantida em memória. Os ganhos durante o processamento da segunda camada, que estaria em uma unidade de disco onde o acesso é mais lento, seriam ainda maiores uma

vez que os nossos algoritmos reduzem o número de entradas a serem descomprimidas, reduzindo assim a quantidade de acessos ao disco.

7.1 Trabalhos Futuros

Apesar dos grandes ganhos em desempenho atingidos pelo BMW-CS, este não garante que o conjunto de resposta exato seja preservado. Para trabalhos futuros, sugerimos um melhor estudo sobre alternativas para se preservar o top-k.

Outro caminho a ser explorado é a combinação de evidências para melhoria da qualidade das respostas retornadas pelo sistema. Podemos estudar como os algoritmos podem ser adaptados para outros modelos, como o apresentado por Shan et al. [2012], onde o autor estuda o impacto da inclusão de fontes de informação externas no calculo da similaridade dos documentos com a consulta. Outro modelo onde podemos estudar o comportamento dos nossos algoritmos é o apresentado por Carvalho et al. [2012], onde o autor propõe um modelo de combinação de diversas evidências de informação em um único valor.

Finalmente, também podemos estudar o desempenho dos algoritmos em cenários onde a coleção a ser processada não possa ser armazenada em memória principal. Nesses casos, esperamos que nossos métodos possam ter impacto ainda maior do que os obtidos quando considera-se que os índices estão todos em memória. Entretanto, diversas questões relacionadas a esse cenário, tais como a presença ou não de sistemas de cache, devem ser estudadas.

Referências Bibliográficas

Anh, V. & Moffat, A. (2006). Pruned query evaluation using pre-computed impacts. Em *ACM SIGIR*, pp. 372--379.

Baeza-Yates, R.; Gionis, A.; Junqueira, F.; Murdock, V.; Plachouras, V. & Silvestri, F. (2007). The Impact of Caching on Search Engines. Em *ACM SIGIR*, pp. 183--190.

Baeza-Yates, R.; Gionis, A.; Junqueira, F. P.; Murdock, V.; Plachouras, V. & Silvestri, F. (2008). Design trade-offs for search engine caching. *ACM Transactions on the Web*, 2(4):1--28.

Baeza-Yates, R.; Murdock, V. & Hauff, C. (2009). Efficiency trade-offs in two-tier web search systems. Em *ACM SIGIR*, pp. 163--170.

Baeza-Yates, R. & Ribeiro-Neto, B. (2011). *Modern Information Retrieval*. Addison-Wesley Publishing Company, USA, 2nd edição.

Blanco, R. & Barreiro, A. (2007). Static pruning of terms in inverted files. *Advances in information retrieval*, pp. 64--75.

Broder, A. Z.; Carmel, D.; Herscovici, M.; Soffer, A. & Zien, J. (2003). Efficient query evaluation using a two-level retrieval process. Em *ACM CIKM*, pp. 426--434.

Carmel, D.; Cohen, D.; Fagin, R.; Farchi, E.; Herscovici, M.; Maarek, Y. S. & Soffer, A. (2001). Static index pruning for information retrieval systems. Em *ACM SIGIR*, pp. 43--50.

Carvalho, A.; Rossi, C.; de Moura, E. S.; Fernandes, D. & da Silva, A. S. (2012). LePrEF: Learn to Pre-compute Evidence Fusion for Efficient Query Evaluation. *JASIST*, 55(92):1--28.

Ding, S. & Suel, T. (2011). Faster top-k document retrieval using block-max indexes. Em *ACM SIGIR*, pp. 993--1002.

Jansen, B. J. & Spink, A. (2003). An Analysis of Web Documents Retrieved and Viewed. *THE 4TH INTERNATIONAL CONFERENCE ON INTERNET COMPUTING*.

Leonardi, S.; Becchetti, L.; Anagnostopoulos, A.; Sankowski, P. & Mele, I. (2011). Stochastic Query Covering. *WSDM 2011*.

Liu, Y.; Zhang, M.; Cen, R.; Ru, L. & Ma, S. (2007). Data cleansing for Web information retrieval using query independent features. *JASIST*, 58(12):1884--1898.

Long, X. & Suel, T. (2006). Three-level caching for efficient query processing in large web search engines. *World Wide Web*, 9(4):369--395.

Moura, E. S. D.; Fernandes, D. R.; Silva, A. S.; Calado, P. & Nascimento, M. A. (2005). Improving Web Search Efficiency via a Locality Based Static Pruning Method. *World Wide Web*.

Nguyen, L. (2009). Static Index Pruning for Information Retrieval Systems: A Posting-Based Approach. Em *Proceedings of LSDS-IR, CEUR Workshop*.

Ntoulas, A. & Cho, J. (2007). Pruning policies for two-tiered inverted index with correctness guarantee. Em *ACM SIGIR*, pp. 191--198.

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

Porter, M. (2006). An algorithm for suffix stripping. *Program: electronic library and information systems*, 40(3):211--218.

Risvik, K.; Aasheim, Y. & Lidal, M. (2003). Multi-tier architecture for web search engines. Em *First Latin American Web Congress*, pp. 132--143.

Robertson, S. E. & Walker, S. (1994). Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. Em *ACM SIGIR*, pp. 232--241.

Salton, G.; Wong, A. & Yang, C. S. (1974). A vector space model for automatic indexing. Relatório técnico, Ithaca, NY, USA.

Saraiva, P. C. & de Moura, E. S. (2001). Rank-Preserving Two-Level Caching for Scalable Search Engines. Em *ACM SIGIR*, pp. 51--58.

Shan, D.; Ding, S.; He, J.; Yan, H. & Li, X. (2012). Optimized top-k processing with global page scores on block-max indexes. Em *WSDM*, pp. 423--432.

Skobeltsyn, G.; Junqueira, F.; Plachouras, V. & Baeza-Yates, R. (2008). ResIn: a combination of results caching and index pruning for high-performance web search engines. Em *ACM SIGIR*, pp. 131--138. ACM.

Strohman, T. & Croft, W. B. (2007). Efficient document retrieval in main memory. Em *ACM SIGIR*, pp. 175--182.

Theobald, M.; Weikum, G. & Schenkel, R. (2004). Top-k query evaluation with probabilistic guarantees. Em *VLDB*, pp. 648--659.

Zheng, L. & Cox, I. (2009). Document-Oriented Pruning of the Inverted Index in Information Retrieval Systems. *AINA Workshops*, pp. 697--702.